Performance Optimization of Linux Networking for Latency-Sensitive Virtual Systems

by

James Matthew Welch

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved November 2015 by the
Graduate Supervisory Committee:

Violet R. Syrotiuk, Chair
Carole-Jean Wu
Gil Speyer

ARIZONA STATE UNIVERSITY

December 2015

ABSTRACT

Virtual machines and containers have steadily improved their performance over time as a result of innovations in their architecture and software ecosystems. Network functions and workloads are increasingly migrating to virtual environments, supported by developments in software defined networking (SDN) and network function virtualization (NFV). Previous performance analyses of virtual systems in this context often ignore significant performance gains that can be acheived with practical modifications to hypervisor and host systems. In this thesis, the network performance of containers and virtual machines are measured with standard network performance tools. The performance of these systems utilizing a standard 3.18.20 Linux kernel is compared to that of a realtime-tuned variant of the same kernel. This thesis motivates improving determinism in virtual systems with modifications to host and guest kernels and thoughtful process isolation. With the system modifications described, the median TCP bandwidth of KVM virtual machines over bridged network interfaces, is increased by 10.8% with a corresponding reduction in standard deviation of 87.6%. Docker containers see a 8.8% improvement in median bandwidth and 4.4% reduction in standard deviation of TCP measurements using similar bridged networking. System tuning also reduces the standard deviation of TCP request/response latency (TCP RR) over bridged interfaces by 86.8% for virtual machines and 97.9% for containers. Hardware devices assigned to virtual systems also see reductions in variance, although not as noteworthy.

To my mother who inspired my curiosity

because she could never answer my questions of "What if...?"

To my wife whose unwavering support, boundless love, and endless patience

have buoyed my spirits and kept me going.

To my children, who give me purpose.

ACKNOWLEDGEMENTS

I would like to express my sincere appreciation for my thesis advisor, Dr. Violet Syrotiuk for her support and confidence in me throughout the classes and research. It's truly rare to find professors that take such keen interest in the success of their students and, for that, I am eternally grateful.

Thank you to both of my committee members, Dr. Carole-Jean Wu and Dr. Gil Speyer, who are both great professors and have inspired me to study networking and architecture.

Thank you to Valerie Young, Shrikant Shah, and Scott Oehrlein who have mentored me at various points during the last three years and inspired me to keep studying.

My deep appreciation also goes out to the rest of the crew in the lab for keeping me on my toes and accepting me unconditionally.

And a final thanks to my old friend and mentor, Dr. Kevin Hazen, who taught me that the precision and accuracy of the work are important, but different doesn't necessarily mean wrong.

TABLE OF CONTENTS

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

1.1 Motivation

Containers have recently increased in popularity as a mechanism for deploying applications and increasing hardware utilization. Virtual machines are widely used in enterprise and their uses have grown dramatically as the cloud computing paradigm and network function virtualization (NFV) have expanded [1], [2]. In this context, the term "virtual system" can refer to virtualization technologies including both virtual machines and containers. Growth has been strong in cloud services for good reason as described by Young et al. [3]. In that work, they describe many of the benefits offered by cloud services such as scalability, controlling quality of service (QoS), customization of user infrastructure, cost effectiveness, and simplified access interfaces, almost all of which are enabled by virtualization. In NFV, the functionality that is often embodied in physical devices such as routers, firewalls, and load balancers may be accomplished with a virtual system which increases flexibility and scalability of the network in many cases. In anticipation of this industry transformation, trade organizations have recently sprouted up to promote NFV adoption and accelerate its development. Both the European Telecommunications Standards Institute (ETSI) and Open Platform for Network Function Virtualization (OPNFV) have been formed for that purpose [1], [2]. Developments in containers have also been enabled by tools for process isolation and application portability, inspiring system designers to reconsider how these technologies fit into the virtualization ecosystem.

Studies examining the suitability of virtualization for cloud, NFV, and high performance computing (HPC) applications, often cite low input/output (I/O) performance as a factor limiting overall performance [3]–[5]. These same I/O constraints also limit application scalability in HPC systems due to the high latency of inter-node communication. Latency can be partially mitigated by modifying operating systems to provide more deterministic operation. The Linux Foundation recently announced that they are fully supporting the efforts to promote and advance the development of realtime Linux [6]. The Linux Foundation is the body that guides and sponsors the direction of Linux so their interest comes at an important time due to recent funding difficulties for realtime Linux [7]. The developments and interest in realtime will help improve one of the greatest weaknesses of virtual machines, the latency and jitter of their I/O. The improved determinism of a system running with realtime performance enables applications that were not previously possible in virtual systems. Innovations in realtime Linux and HPC should also improve NFV and cloud performance by making these systems more responsive and predictable.

An important motivation for virtualization is to increase process density which improves hardware utilization, bringing down costs. Density of processes and applications may be increased even further by utilizing containers such as Docker [8]. Although containers have potential to scale to hundreds or thousands of containers per host, these special cases are often not possible without specific homogeneous workloads. Memory contention and I/O density are often limiting factors in system performance such that process density can not increase without addressing memory and I/O limitations. Containers can very efficiently share and utilize available CPU and memory resources, but scalability is limited for containers that need high bandwidth or low latency network communication.

1.2 Contributions

Many studies have analyzed virtual machines and containers to compare their performance [3], [4], [9]–[18]. These studies have performed valuable analyses of virtual systems and provided significant insight into their performance issues. These studies are commonly in the context of cloud computing, NFV, or HPC. Network performance is important in these infrastructures due to the cooperative nature of the network in a large-scale system. These studies frequently find that I/O is an issue, yet no modification is suggested by their authors to improve its performance. The systems analyzed in those studies differ little from their "factory defaults", which produces results that ignore real-world deployments and goals such as maximizing performance.

Tuning to improve performance can be a challenging task due to poorly-defined endpoints and deeply-connected dependencies between options which complicates assessing the benefit of any particular choice. This is further complicated by the nondeterministic nature of operating systems which may hide the effects of some tuning. Many system parameters, however, have well-known benefits or penalties such as addressing known bottlenecks in memory and I/O. Improving the performance of I/O is, however, complicated by difficulties that arise from sharing peripherals and scalability problems that accompany bridging and similar networking paradigms.

The results of previous comparisons are built upon here by demonstrating a straightforward, logical tuning process that addresses CPU, memory, and I/O performance enhancements to improve network performance. Performance improvement is accomplished through patching the host and guest kernels with the `preempt-rt` patchset [19]. The tuning and `preempt-rt` patching mentioned above are applied to the Linux `3.18.20` kernel to produce the `3.18.20-rt18 PREEMPT-RT` kernel which is then built with specific configuration choices to enhance performance. The `preempt-rt`

kernel is further tuned by applying specific boot options to improve its process isolation and virtual memory performance. The `preempt-rt` tuned kernel improves the determinism of both virtual machines and containers. The Linux kernel is the main focus of tuning here without modification of hardware parameters to provide benefits that can be realized on the majority of modern hardware.

This work compares the performance of the two kernels running popular open-source virtual systems including kernel virtual machine (KVM) and Docker [8]. No feedback-driven optimizations are performed in favor of making informed, logical choices to improve performance. The network performance of bridged and physical network interfaces is measured using common open-source tools such as `netperf`. Testing completed with `netperf` includes measurement of TCP and UDP bandwidth, along with TCP, UDP and ICMP latency.

This thesis demonstrates that modifications to the kernel can improve network determinism significantly. One result of note is standard deviation in TCP bandwidth decreased as much as 97% for containers using bridged networking. In some cases, system tuning and process isolation can increase performance as well, but the primary goal of this work is to reduce variance in bandwidth and latency. The results presented here show that realtime tuning of the kernel may offer greater benefits to shared networking paradigms such as virtual bridges then physical network interfaces. This is a because the improved determinism of the realtime kernel benefits kernel-dependent processes like bridging whereas physical interfaces often use direct memory access (DMA) and CPU virtualization extensions which already limit operating system involvement to improve performance.

1.3 Organization

This thesis is divided into five chapters, the remainder of which are organized as follows. Important related work that has enabled performant virtual systems is described in Chapter 2 along with some discussion of the previous performance analyses comparing virtual systems. Virtual machine and container architecture are also briefly discussed in Chapter 2 to highlight areas for improving performance. Test platform hardware descriptions and a discussion of the operating system modifications made are presented in Chapter 3 with motivations for those choices and their intended effects. The experimental procedures and protocols including the measurement process and types of network tests performed are also discussed there. Results and discussion of the network performance measurements described in Chapter 3 are presented in Chapter 4 along with their significance and implications. Chapter 5 discusses the implications and conclusions from the measurements collected here. Potential directions for future work that resulted from the investigation performed here are also presented in Chapter 5.

Chapter 2

RELATED WORK

There have been many comparisons between virtual systems, but they often do not use any system optimizations to improve network performance of the virtual systems under test. It is possible to improve the network performance of virtual systems with fairly simple configuration changes to the kernel and modification of boot parameters. This chapter discusses some of the enabling developments in virtual system architecture as well as some of the comparisons among them.

2.1 Virtual Machine Architecture

In this section, virtual machine architecture is discussed to demonstrate resource needs and potential areas for improvement. Differences among hypervisors and variations of virtualization are then discussed, followed by an analysis of innovations in CPU, memory, and I/O virtualization.

The architectures available in virtual environments are as widely varied as the types of hardware they are capable of emulating. Many of the concepts currently in use regarding virtual machines originated in a seminal 1974 paper by Popek and Goldberg [21]. Although virtual machines had already been implemented on "third-generation computer systems" such as the IBM 360/67 by then, Popek and Goldberg sought to establish formal requirements for and prove whether other systems of that era were capable of supporting a virtual machine monitor (VMM)[21]. At the time, their analysis focused on the possibility of a VMM, but the term hypervisor has largely come to replace VMM as the name of a software system that allocates, monitors, and controls virtual machine resources as an intermediary between the hardware and the virtual

machine's operating system (OS). What follows in this section is an introduction to some important concepts in hypervisor and virtual machine architecture and the work that led to them.

2.1.1 Hypervisors

Hypervisors are available in a few flavors, depending on the where the hypervisor is running in relation to the hardware and the guest operating system. The term "guest" here is used to refer to a virtual machine that is running on emulated or hosted hardware and the term "host" is used to refer to the hardware, native operating system, and hypervisor. In a Type 1 hypervisor, the hypervisor process is executing directly on the CPU or "bare metal". That is, the hypervisor code is not being hosted, translated, or controlled by another system or piece of software, but running as a native process on the CPU. In contrast, Type 2 hypervisors require a host operating system to provide some system services including memory and file system management.

The difference between these two types of hypervisors is that a Type 1 hypervisor does not require an operating system to run, whereas a Type 2 does. Although Type 1 hypervisors do not require an operating system to *run*, they do require an operating system for *control* of the hypervisor and guests [22]. Type 1 systems appear to have an efficiency advantage over Type 2 since they tend to use microkernels instead of the macrokernels that are usually required to host Type 2 hypervisors. According to Liguori [22], however, the difference between them has little to do with performance, robustness, or other qualitative factors, but, rather, relates back to observations about their differences made by Popek and Goldberg [21] and the analyses of these differences by Robin and Irvine [23]. Robin and Irvine's analysis related to the potential for the Pentium processor to support a secure VMM [23], so it was important to draw conclusions about the capabilities and suitability of various hypervisors. The vendors

of hypervisor solutions also have an interest in drawing distinctions between the two types of hypervisors, but the need to differentiate between these products has become less important over time.

Although the distinction between Type 1 and Type 2 hypervisors has narrowed, the market for virtualization is full of both. VMware, one of the more popular enterprise virtualization vendors, has products covering both architectures including ESXi, their Type 1 enterprise hypervisor product that is responsible for running large-scale virtualized systems all around the world [24]. They also provide Type 2 hypervisors such as VMware Workstation Player that runs on systems from desktops to small-scale servers that do not need the higher levels of orchestration and performance offered by ESXi.

In addition to numerous other available hypervisors, the Kernel Virtual Machine (KVM) is the default hypervisor in Linux that has been included as a module in the Linux kernel since February 2007 when it was included with Linux kernel version 2.6.20 [25]. The KVM module allows the host operating system to provide the low-level hardware access to the guest that has been enabled by hardware virtualization extensions such as Intel's VT-x [26]. Along with KVM in Linux, there is a user-space component known as the Quick EMUlator or QEMU[27]. QEMU was designed to be an architecture-agnostic emulator and virtualization helper, capable of running software written for one architecture on other architectures. QEMU can achieve relatively good performance using binary translation, but, when paired with KVM or the Xen hypervisor [28], it can achieve near-native performance by executing guest instructions directly on the host processor.

Combined, the KVM/QEMU hypervisor runs natively on Linux with kernel modules and userspace tools, but additional libraries such as `libvirt` [29] can be utilized to simplify VM creation, monitoring, and orchestration.

2.1.2 CPU Virtualization

In addition to the variations in hypervisors, the type of virtualization used to provide devices to the guest operating system can have a significant impact on performance. In the following sections, some important types of hardware emulation are reviewed comprising binary translation, paravirtualization, and hardware assisted virtualization.

One of the early challenges in the virtualization of the x86 architecture was handling privileged instructions. The x86 processor was originally designed with 4 "rings", numbered from 0 to 3, which represent decreasing privilege levels as the rings increase. Operating systems utilizing the x86 instruction set execute user code in ring 3 and privileged instructions (kernel code) in ring 0. Virtual machines running on the x86 architecture execute their instructions as a user-space process in ring 3 so the host OS can maintain control of the system. The mechanism by which the processor executes privileged instructions on behalf of the guest has been the topic of considerable effort in the advancement of virtualization which has spawned multiple techniques for handling these instructions.

One of the earliest methods, known as binary translation, involved trapping privileged guest instructions in the hypervisor and translating them into "sequences of instructions that have the intended effect on the virtual hardware" [30]. The binary translation technique, developed by VMware, was one of the most efficient methods in early hypervisors, but is now considered to have high overhead due to the additional time required to perform the code translation. This same translation process, however, allows a hypervisor using this technique to run guest operating systems for virtually any processor on any other instruction set, provided that an efficient translation can be achieved. This flexibility makes binary translation one of the most versatile methods

of virtualization available and well-suited for virtualization of old instruction sets or hardware.

Paravirtualization is another technique for handling guest privileged instructions. It involves cooperation between the guest and host operating systems to improve efficiency. The guest OS must be modified to replace privileged instructions "with hypercalls that communicate directly with the virtualization layer hypervisor" [30]. VMware has incorporated this method in their vmxnet series of network drivers to accelerate network workloads. A more widely known example of paravirtualization is one of the first open-source hypervisors, known as the Xen hypervisor [28]. At the risk of oversimplification, the Xen project is a modified Linux host that communicates directly with the guest kernel. The guest kernel and modules must also be modified in a paravirtualized system to facilitate this communication which places an additional burden on hardware vendors to provide not only open source drivers but also paravirtualized open source drivers for their hardware. Although the Xen hypervisor was originally developed with the intent of being hardware agnostic, the modifications required to the guest operating system mean that only vendors wishing to participate in the open-source community provide paravirtualized drivers. The community itself is free to develop these drivers, but this adds a barrier to adoption for most new hardware products. Despite the additional effort required, Xen maintained their lead as a very popular open-source hypervisor for many years. Paravirtualization has its fans, however, and much work has been done comparing Xen to containers and other hypervisors such as KVM [3], [14]–[18], [31].

The third popular virtualization method is becoming the *de facto* standard as virtualization matures. Hardware assisted virtualization started out as an effort to accelerate instruction translation, but early generations of hardware had difficulty keeping up with the binary translation method preferred by VMware [30]. The silicon

vendors, however have been improving their virtualized performance and adding hooks to enable virtualization of their hardware [26]. Since operating systems for x86 were already common when hardware assisted virtualization was introduced, the architecture needed subtle modifications to help enable virtualization without breaking existing software. This was accomplished with the introduction of yet another protection ring that runs below the kernel's ring 0. The hypervisor runs in ring -1, below the operating system kernel, and traps privileged instructions when they are executed by the guest. For each virtual machine, a new structure is created and maintained in the host kernel memory. Logically similar to a process control block, this structure, is commonly known as a Virtual Machine Control Structure (VMCS) or, alternately as a VM Control Block (VMCB). The VMCS maintains the state of the virtual machine in the host kernel and is updated when the guest needs to perform a "vm-exit" to allow the host to execute privileged instructions. This vm-exit is the process of a virtual machine preserving its CPU context then returning control of the CPU to the host OS and has been a significant source of latency for guest privileged instructions. Advancements in virtualization-aware driver models such as SR-IOV, however, have improved both latency and the frequency of these exits [32], [33].

A deep discussion of the hardware systems and mechanisms of processor and platform virtualization outside the scope of this thesis. Suffice it to say, however, that the CPU vendors have significant interest in producing higher core-count CPUs and supporting virtualization. The discussion of these processors is limited even further to x86 architecture, but it should be noted that ARM and other vendors are actively working to enable virtualization with similar methods. Additionally, important components of virtualization such as SR-IOV are managed by the Peripheral Component Interconnect Special Interest Group (PCI-SIG) and are not the exclusive domain of

x86 architecture [33]. To that end, both Intel and AMD have independently developed processor extensions to enable virtualization [26].

2.1.3 Memory Virtualization

Virtualization of the CPU was the first challenge in the development of secure virtualization, but, in Von Neumann processor architecture, the memory unit is equally important. Memory has become an increasingly significant source of system latency as the performance of the CPU core has improved faster than memory performance, so improvements in this area are doubly beneficial for virtual machines. Virtualization uses memory structures conceptually similar to those developed for virtual memory in early computing systems. Virtual memory was created to allow multiprogramming and process address spaces that are larger than the available physical memory. Modern CPUs implement virtual memory with the aid of a memory management unit (MMU) and translation lookaside buffer (TLB) to manage page tables and accelerate page lookups. Fairly recent developments in x86 processors have allowed hypervisors to maintain guest memory mappings with shadow page tables, known as Extended Page Tables (EPTs) in Intel processors and Nested Page Tables (NPTs) for AMD. A hypervisor may use TLB hardware to map guest memory pages onto the physical memory similar to a native process, reducing the overhead of guest OS memory access.

Another significant improvement to virtual machine memory performance can be realized by utilizing hugepage memory [34]. Hugepages, referred to as superpages by Romer et al., can be described as a memory page that is a power-of-two multiple of a standard 4096 byte (4 kb) memory page. Romer et al. demonstrated a significant improvement in performance when using hugepages for memory-hungry applications. In systems with relatively small memory sizes, virtual memory and swapping smaller pages was more efficient than larger pages. As memory sizes have grown to hundreds of

gigabytes per server and applications can consume multiple gigabytes each, however, 4k memory pages no longer seem an obvious fit. Hugepages seek to reduce the frequency of TLB lookups and page table walks by using larger pages of memory. With respect to performance, they showed that TLB overhead could be reduced by as much as 99% using *superpage promotion* which is a system of aggregating smaller pages together as they are used by a process. In contrast, current implementations of hugepages in Linux offer a set of large memory pages available to the kernel for memory allocation. Unlike superpages, hugepages must be allocated at boot, rather than being coalesced dynamically, but Linux hugepage performance is similar aside from increased memory consumption due to unused portions of hugepages. For virtual machines that are potentially using multiple levels of memory page walk, any reduction in the frequency of lookups is beneficial. Hugepages are included here as an important mechanism for improving host and guest memory performance.

2.1.4 I/O Virtualization

A recent development in full system virtualization is the virtualization of peripheral devices and I/O (IOV). Along with compute and storage, processing of I/O is one of the most critical components of a server's workload since I/O can be a large source of latency for remote transactions. Efficiently utilizing I/O allows virtual machines to perform tasks that were previously possible only in native systems, particularly the processing of network packets and workloads. Similar to the "virtualization penalty" that occurs with nested page table lookups and binary translation, latency inherent in the processing of network packets with multiple levels of handoffs should be avoided when possible. Virtualization is essentially software emulation of hardware devices so the natural first attempt at device virtualization is to emulate a device in the kernel, providing a software device to the guest OS that is based on a common physical device.

This method is common in some workstation virtualization products such as VMware workstation [35].

Instead of emulating a software device in the kernel, it is also possible to emulate the device in user-space. This is the method used by QEMU which provides both device emulation and a platform-agnostic hypervisor. In Linux operating systems, QEMU is often combined with the KVM modules to enable full system virtualization. In this configuration, QEMU provides user-space device emulation for simple devices such as mouse and keyboard, while KVM provides virtualization of the physical hardware. Userspace emulation has the advantage of removing the responsibility from the kernel, thereby minimizing the potential attack surface of the kernel. As mentioned earlier, paravirtualized devices are another variation on this theme of emulation where the paravirtualized guest drivers communicate with the host. In addition to Xen's paravirtualized driver model, the KVM `virtio` library is the basis for paravirtualized devices in Linux [36]. The `virtio` library uses QEMU to implement the device emulation in userspace so host-side drivers are not necessary.

While device emulation can provide important flexibility and hardware independence, it brings up the recurring theme of software managing hardware functions which is often less efficient than utilizing dedicated, specialized hardware to perform the function. The alternative is to avoid any device emulation and allow the guest OS to access hardware directly as if it belonged to the guest rather than the host system. Since it is conceptually similar to virtualization of the MMU for memory, virtualizing the direct memory access (DMA) transactions of modern I/O devices requires an I/O MMU (IOMMU) to allow communication between the guest OS and I/O devices. In the x86 architecture, this feature is known as AMD-Vi or Intel VT-d, but both utilize the same concept of an IOMMU to avoid vm-exits when processing I/O. This allows the hypervisor to unbind a hardware device from its kernel and "pass through" or

"direct assign" the device to the guest OS [35]. Direct assignment of hardware to guests also comes with the cost of dedicating network interfaces or other important devices to a guest, but these devices provide considerable performance improvements over paravirtualized or emulated guest devices, thereby enabling applications that could not previously be virtualized due to low-latency constraints.

If a host system has a large number of virtual machines that need high-performing I/O, it can be difficult to fit enough peripheral cards into the host chassis to serve as passthrough devices to the guest VMs. A potential solution to this resource conflict can be found in PCI-SIG Single Root I/O Virtualization (SR-IOV) [33], [37]. SR-IOV is a general method by which the host system can configure a single hardware device to create and control multiple additional *virtual functions* of the device. These virtual functions can be directly assigned to and accessed by the guest, similar to passthrough, without removing the main physical function from the host. The host's physical function represents the original hardware and is responsible for the management functions of the peripheral. An illustrative example is the Intel 82599 10 Gigabit network card, used later in this study. Each physical network interface (physical function or pf) in these network cards has 64 Rx/Tx queue pairs that are normally used as send and receive buffers to maintain multiple simultaneous flows. The individual queue pairs may be "broken out" of the main pool to create a virtual function that is, essentially, a new network device sharing the same physical interface as the physical function. These new devices are assigned unique MAC addresses and IP addresses to differentiate them on the network so that an outside observer cannot tell them apart from a physical device. The advantage with SR-IOV is that one physical interface can be multiplexed into independent interfaces that are each able to reach near-native performance levels [32].

2.2 Containers

At a high level, containers can be viewed as another level of access control beyond the traditional user and group permission systems. While those systems provide resource access control, containers can allocate these resources with finer granularity, thus exposing only those resources and privileges that are required by the process [14]. Containers are capable of applying controls to program execution that should be included in process management at a fundamental level.

2.2.1 Container Architecture

Also known as *operating system virtualization*, containers are a construct of the operating system that serves to provide limited context and resources to a process executing on the host system. One of the most important duties of the kernel is to control access to the system's resources including CPU, memory, and I/O. All processes under OS control should have a view of the system that is limited in context and scope – they should only have access to the resources that they require with some margin for error. The Linux subsystems that enable a containerized process, including `chroot`, `cgroups`, and `namespaces`, have been gradually added to the Linux kernel over time.

2.2.2 Resource Scope and Control

An early implementation of process isolation that led to containers was the restriction of its file system view, embodied in the `chroot` system call and program. Introduced in Version 7 Unix in 1979, the `chroot` system call was later expanded into BSD Jails and Solaris Zones [38]. This tool allows a user to change the root directory of a process and its children to a specified directory such that the process

subsequently views that directory as its root directory, `/`. This mechanism is useful for providing partial file system isolation to a process, thus controlling libraries and binaries that are available to that process. Although this works well for a normal process, it does not prevent a malicious process from accessing arbitrary files, so it can be easily circumvented. It makes this process more difficult, thereby "hardening" the target, but was not intended to fully sandbox a process or restrict its file system system calls.

The next process isolation tool to become available in Linux was control groups (`cgroups`). Cgroups are organized into hierarchies with the root being the default `cgroup` for all processes. Contributed by Google [39], `cgroups` serve to limit the resources available to a process in a hierarchy, enabling child processes to inherit their parent's `cgroup` assignment [40]. Each process must belong to one and only one `cgroup`. Child processes can be organized into sub-trees that subdivide the resources of its parent process. In addition to isolation, `cgroups` are also capable of partitioning resources so that a process may be allocated a proportion of a CPU's resources and share the remainder with other processes or not. The set of `cgroups` available includes `devices`, `hugetlb`, and `systemd`, among others; the tuning performed here only utilizes the `cpuset` and `memory` hierarchies for process isolation.

`Namespaces`, also a recent addition to the kernel, are the mechanism by which the scope or resource view of the process may be limited and controlled. The `namespaces` currently available in the Linux kernel are `pid`, `mnt`, `net`, `user`, `ipc`, and `uts`. In this work, the only `namespace` that was explicitly invoked by the user was the `net` namespace. Docker's typical configuration places the container processes inside a namespace that is specific to the container. Importantly, this sets the scope of various container subsystems including: processes running on the system (`pid`), the file system mounts (`mnt`), network interfaces (`net`), users (`user`), inter-process communication

(ipc), and hostname (uts). Although Docker does not currently support assigning physical devices to containers, this study accomplished this task with the iproute2 library and its ip user space tool.

Linux "capabilities" are the mechanism by which the monolithic permissions granted to the root user for full system administration are broken down into more granular permissions. Rather than assigning full system permissions to each container, the Docker daemon uses capabilities to deliver a more finely grained set of permissions. Any capability may be assigned or removed individually at run time, depending on the needs of the container process. An important example is the capability required for chroot calls, logically called cap_sys_chroot, which is seldom assigned to containers unless they are intended to have deep system access.

2.2.3 Docker Networking

Docker networking can be fairly complex with multiple options available to both the daemon and the run tools [41]. There are a few standard varieties of network interfaces that are commonly used in containers. Most common among software options, virtual bridges are operated by the host kernel with a non-trivial CPU involvement. Software bridges are also commonly used by virtual machines where they are combined with the virtio library to provide a paravirtualized interface to guest VMs [36]. Bridges are the default layer 2 networking used by Docker which enables the daemon to quickly configure simple network topologies for container cooperation [41]. The Docker daemon also uses the common Linux firewall iptables for layer 3 connectivity, routing, and network address translation (NAT) to allow containers to communicate with the outside world. Another reason to study the bridged networking paradigm is that bridges are generally important in networking and useful in interesting new system topologies where containers may be linked together or their functions pipelined.

2.2.4 Container Systems

While Docker has recently popularized process containers, related concepts have been used in earlier operating systems. FreeBSD introduced the Jails concept in 1999 as a process isolation technique [38]. The Solaris version of the Unix operating system added the concept of Zones (Solaris Containers) in 2004, as an upgrade of the BSD Jails concept. Other container systems have been developed since including the popular Linux VServer and OpenVZ, but these systems are not native Linux containers, the focus of this thesis [5], [15], [16]. LXC is the current default Linux container standard, added to Linux as an attempt to bring containers into the mainline kernel [42]. The Linux container infrastructure has evolved some as have the many other container systems to the point that there was need to unify the standards on a single container format, `libcontainer`, which has recently been adopted by the Open Container Initiative under the Linux Foundation [43].

The Open Container Initiative (OCI) was formed as a collaborative project under the Linux Foundation [43]. Members of this organization span the industry from major cloud players such as Microsoft and Amazon to virtualization vendors such as VMware and CoreOS. The goal of this new organization is to promote an open, standardized format for containers and the engine runtime libraries. They have a head start on the standardization process with Docker donating its container format, `libcontainer`, and runtime, `runC`, to the organization as a starting point. The development of a common container and runtime for Linux under the OCI should accelerate the standardization and adoption of containers under Linux. As a consequence of the OCI, Linux containers should eventually converge onto a single format, which has evolved from the Docker container specification.

The Linux subsystems supporting containers have become sophisticated enough that Docker or LXC are no longer strictly necessary to create containers, because administrators can create their containers with native Linux tools such as `systemd` or scripts that stitch together the correct `cgroups`, `chroot`, and `namespace` allocations. Based on the amount of media coverage and hype surrounding the young startup, Docker has become the *de facto* standard for containers in Linux. Docker is popular for good reason: in addition to providing a robust tool set for isolating processes and container orchestration, the Docker team has been extremely open about their development and responsive to issues submitted by users on their `github` account [44]. It is for the reasons of its ubiquity and ease of use that Docker is the container format selected for comparison in this thesis.

2.2.5 Docker Ecosystem

Docker is based on the underlying Linux process isolation systems combined with user space tools such as `docker daemon`, `docker build`, and `docker run` written in Google's Go language [45]. Docker aims to enable process isolation in containers with simple, effective tools, but the ecosystem of Docker is what makes it so useful. In Figure 2.1, the overall flow of a container image is shown as it moves from registry to daemon and is then deployed as a live container, all controlled through the client system. In this figure, the "DOCKER_HOST" and "Client" systems are shown as separate entities, but they are all run on the same host system. A registry in this context is a repository of container images that are posted and curated by their authors. This system allows a Docker client to query the daemon for the presence of a particular image, organized by hashes, similar to other version control systems. If the daemon does not have the requested image in its stores, it can query official registries for the image and download a copy if available. Once the image is present in the storage pool,

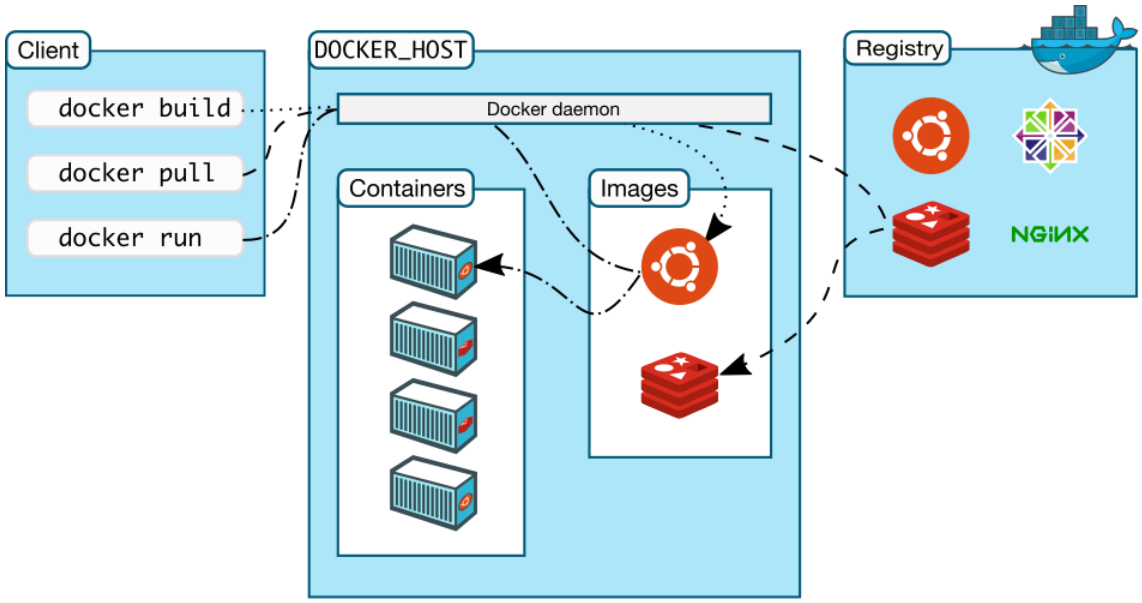

Figure 2.1: Docker Ecosystem Architecture [45]

the daemon can then launch the image as a container or build new images based on it. Additional development or enhancement to that container may then be committed on top of the image and shared with others through the registry or just exporting the image as a compressed tarball of its filesystem. This version-controlled sharing of official images enables rapid development and deployment of containers and has contributed greatly to the popularity of Docker as an application deployment tool.

2.3 Comparisons and Performance Analysis

Some of the important architectural details of virtual machines and containers are described in Sections 2.1 and 2.2. An interesting quote on the differences between virtual machines and containers comes from [13]: "Hypervisors abstract hardware, containers abstract the operating system". This concisely summarizes one of the important differences between these two types of virtualization in that virtual machines must emulate hardware devices with software which often has some performance impact. Virtualization of the operating system, however, only creates minimal additional

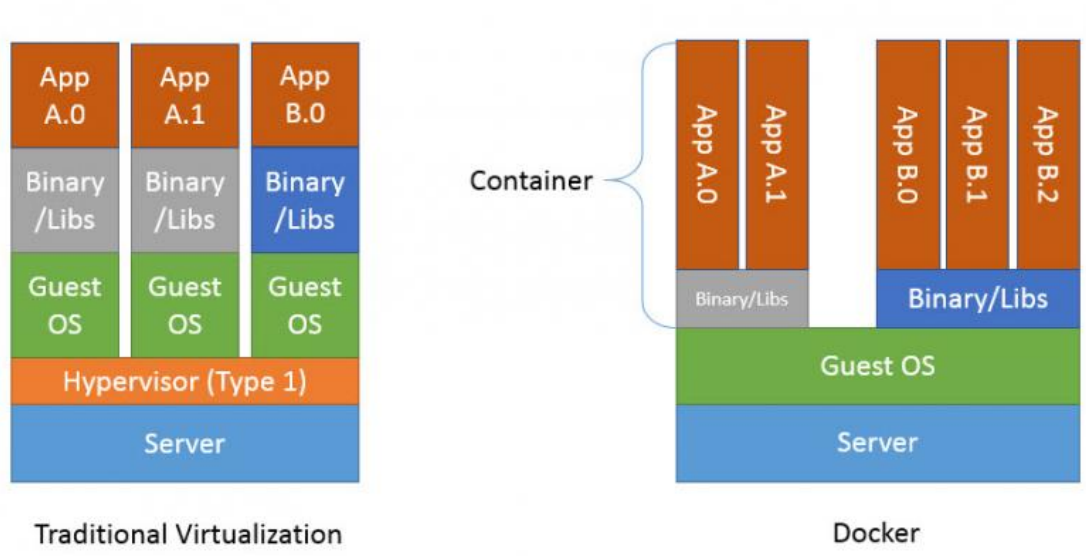

Figure 2.2: Architecture Overview of Virtual Machines and Containers [46]

overhead with containers, most of which are felt during setup rather than runtime. Figure 2.2 provides a simplified comparison of their architecture. In this traditional virtualization schematic, the hypervisor sits on top of the server operating system and hardware. It is responsible for mediating access to the host hardware as well as emulating devices for the guest. This additional hardware emulation comes with some performance cost as does the hosting of a complete operating system for each guest. Although the Docker system in Figure 2.2 shows the guest OS running on top of the server OS, containers are not emulating another OS, but utilizing the host's OS with limits imposed by the host and container orchestration system.

2.3.1 Performance Analysis

Many studies to date have compared the performance of virtual machines and containers. Virtual machines have multiple additional libraries and operating system instances, which increases the latency of protected operations.

Felter et al. made an extensive evaluation of containers and virtual machines including memory, compute, and I/O, but they did not attempt to tune the systems [14]. They also used the advanced multi layered unification filesystem (`aufs`) as the Docker storage driver, which is deprecated in upstream kernels.

In an excellent comparison between containers and virtual machines, Scheepers analyzes Xen and LXC with macro benchmarks [16]. He showed that public clouds need virtual machines for their isolation potential, but containers can be used to maximize the *utilization* of the hardware.

Wang et al. studied the performance overhead of dynamic resource allocation and found that static allocation performed better in most cases [17]. The performance degradation of Xen virtual machines was attributed to intensive network I/O. This degradation could be addressed in part with static allocation of network interfaces and physical device assignment. They concluded that decreasing the switching magnitude and frequency of their allocation schedule led to performance improvements. At the lower limit of switching frequency is static allocation of resources, with processes pinned to CPUs and dedicated physical resources.

Sherry et al. examined middleboxes which are the heart of NFV [47]. They show that, as demand for streaming data increases, the need for additional middleboxes to process and handle the traffic is also increasing. These network functions are often allocated to virtual machines due to the flexibility and scaling that can be achieved with increased density.

Middleboxes are also examined in a paper by Sekar et al., where they make the case that middlebox innovation, i.e. in the context of NFV, is at least as important as for switches and routers [48]. They propose that the role of middleboxes in the network includes the critical usability and security considerations of network design. One idea presented in their article that has significant potential with containers is the

chaining of middlebox functions and its analogy to processing operations on a packet. This is one of the core ideas of NFV in that the entirety of the network functions need not be accomplished in one system or even one physical box, but can be distributed around the network as processing nodes. By their estimates, there also appears to be a significant amount of common traffic overlap ranging from 64 to 99% in middlebox processing. While this seems high, it also presents an interesting alternative use case for containers and virtual machines which may be pipelined and even potentially share resources to decrease the redundancy present in current schemes.

Xavier et al. examine the suitability of Linux vServer, OpenVZ, and LXC for high performance computing environments [4]. While the high compute-density of containers is an advantage in HPC, it is often the network that has the greatest impact on system performance. Containers lack the necessary isolation for co-located workloads, but, since HPC workloads and cluster topologies are often controllable and predictable, the lack of isolation in containers should not pose a significant impact to performance. One of the shortcomings of their study, however, was not using high-throughput network interfaces that are so critical in HPC, which makes it difficult to properly draw distinctions among the systems. That comparison, specifically since it was targeted at HPC, would have benefited from more CPU isolation and tuning.

The work of Wang and Ng analyzed the impact of virtualization on network performance in the Amazon public cloud [31]. Although that work found that Amazon does not have the same emphasis on performance as an HPC cluster, there are instances available in the Amazon cloud that completely isolate their CPUs for guaranteed performance. Although Wang and Ng did not have access into Amazon's orchestration or control systems, it is improbable that the high-performing virtual machines were oversubscribed with other systems sharing their CPUs. These high-performing virtual

systems are the subject of this thesis and it is the goal of the performance tuned kernel described in Section 3.1.3.

There is a significant interest in utilizing virtual systems for HPC. Another study that investigated this paradigm was performed by Younge et al., where they performed an in-depth analysis of current virtualization technologies in HPC [3]. They investigated a set of current hypervisors for performance which helps to guide the choice of hypervisors in this thesis. In their article, they examine VMware, KVM, Xen, and VirtualBox hypervisors. VMware has previously required permission prior to publishing which complicates using their software in academic research. The proprietary nature of their software also limits its utility in work that seeks to better understand their operating system. The popular Xen hypervisor, while open source, requires paravirtualization which creates additional burden. VirtualBox, while widely available, is also not open source and has little available performance data. KVM is selected due to its ubiquity and deep library of literature supporting it. This choice is also supported by [3] who also proclaim it the best performer in internode bandwidth.

Rathore et al. compared KVM and LXC performance and isolation in hardware-assisted virtual routers, but used dynamic allocation which normally has a negative impact on latency [15]. Their isolation could have improved with realtime tuning and CPU isolation. Containers alone cannot provide real isolation of processes without explicit CPU control with `cgroups`, but the organization of processes and CPU assignment can be tricky. Orchestration of containers and virtual machines needs to consider process isolation and CPU over-subscription for maximal I/O performance.

One of the considerations when designing the performance analysis of network systems is the composition of the traffic that they handle. Virtual systems often reside in data centers where it has been reported that TCP traffic represents a share of about

90% if the overall traffic [49]. This observation highlights the need for performant network interfaces due to the high overhead required in maintaining TCP flows.

While many of the performance analyses described here are focused on high I/O performance for virtual systems, they all neglected to investigate the impact of kernel configuration or static resource allocation to improve the network performance of those systems. In Chapter 3 to follow, the impact of kernel performance tuning and different types of network interfaces are analyzed. The analyses performed show significant improvement to TCP latency in most cases and bandwidth in some.

The next chapter describes the experiments performed and some of the motivations for design decisions. The platform used for the experimentation is described along with the procedures used to collect measurements of system performance.

Chapter 3

EXPERIMENTAL DESIGN

As described in Section 2.3.1, the performance comparisons that have been made between containers and virtual machines have, for the most part, not tuned their systems for the comparison. Tuning can present a significant challenge due to the numerous parameters and the resulting exponential number of configurations possible. To avoid this potential explosion in the scope of the comparison, performance analyses are restricted to two system configurations and two network types for each type of virtual environment. The two configurations include the standard Linux kernel, version 3.18.20, and the same kernel with `preempt-rt` patches applied [19] and some important tuning parameters for the kernel and environment. Network interfaces compared include the standard bridged networking for each virtual system and a physical interface in passthrough to a VM or assigned to the namespace of a container.

3.1 Experimental Setup

In this section, the experimental environment constructed for this study is described comprising system hardware, operating system versions, kernel versions, kernel parameters, kernel configurations, Docker parameters, and QEMU parameters.

3.1.1 Hardware

The motherboard used for the test system is a SuperMicro X10DRH-C/i server board with dual CPU sockets. Both CPU sockets on the motherboard are populated with an Intel Xeon E5-2608L v3, having 6-cores running at 2.00 GHz. Hyperthreading, CPU Turbo, P-states, and C-states are all disabled in the BIOS as a performance

optimization [19]. The platform is populated with two 16 GB memory sticks per socket for a total of 64 GB of DDR4 memory running at 1866 MHz in dual-channel mode. The network card used is an Intel 82599ES 10-Gigabit PCIe card with two optical interfaces, each directly connected to the client system. The 10 Gigabit network interfaces are used in order to stress the performance of the virtual systems. The computation involved in processing a 1 Gbps flow is not significant enough to stress a virtual machine, whereas a 10 Gbps stream provides a significant enough workload to observe significant differences between the systems.

3.1.2 Networking and Topology

As discussed in Section 2.2.3, the default networking paradigm of Docker is to create a bridge and pair of virtual Ethernet interfaces [41]. Both interfaces are bound to the bridge; one remains in the host's namespace and the other is assigned to the container namespace as its primary network interface. Figure 3.1 shows the topology of the network connections to virtual systems as configured for this study. Each virtual system has three network interfaces including localhost (127.0.0.1, not shown), a physical interface, and a bridged interface. Shown in green in the figure, the physical interfaces to virtual machines utilize VT-d extensions and the CPU's IOMMU to provide direct access to the hardware, but containers only need the physical interfaces placed in their namespaces. Represented by red in the figure, the bridged interface used for both virtual systems traverses a standard Linux bridge provided by the `bridge-utils` package in Ubuntu and Debian. The virtual machine bridge differs slightly from the container in that the guest OS uses the `virtio` paravirtualized interface provided by QEMU. Each network test performed in this study has the server inside the virtual environment and the client running natively on an adjacent, similar test system.

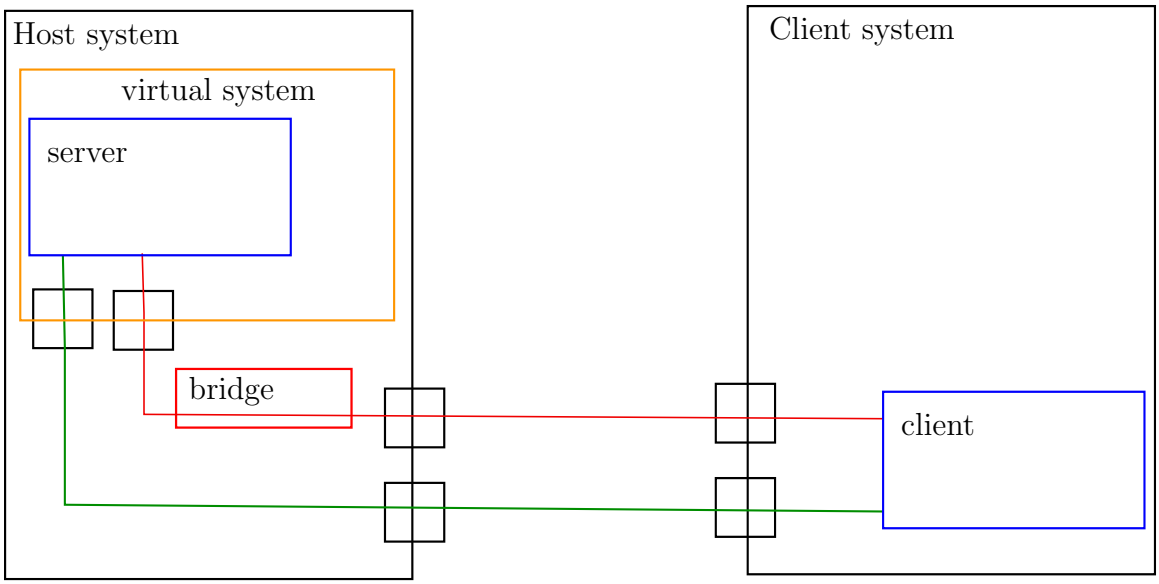

Figure 3.1: Topology of Experimental Network

Linux `iptables` is also used for Docker networking on the host since it represents the default networking configuration of Docker. No effort is made to optimize the performance of the bridge or `iptables` because the focus of this work is not on tuning the network interfaces themselves, but investigates how a latency-sensitive system can be configured to improve performance.

In order to improve performance for both virtual machines and containers, a physical interface may be passed from the host into the domain of the virtual machine or into the namespace of the container. As described in Section 2.1.4, this is known as passthrough for virtual machines and is supported by CPU virtualization extensions [26]. In the version of Docker used, 1.8.1, direct physical assignment is not enabled by default. Instead, a script is constructed to assign a physical interface to the container namespace, with inspiration borrowed from Petazzoni's pipework [55], Anderson's direct-phys fork of pipework [56], and a serverfault.com post [57].

3.1.3 Software Environment

The operating system used in this study needed to have a balance of stability, advanced features, and performance, so a Long Term Support (LTS) version of Ubuntu, 14.04.3 is chosen. The Linux 3.18.20 kernel is selected for both the host and guest OS due to a set of factors affecting usability and performance. Choosing which Linux kernel version to use for this study is complicated by factors that are not known until system configuration. It was already understood prior to configuration that Docker is not supported in kernel versions prior to 3.10 which aided in limiting potential kernels. One of the secondary goals of this study is to use as much standard and open software as possible so that the kernel and system configurations could be easily reproduced by other investigators. This goal influenced the choice of kernel due to the performance impact of the Docker's storage driver.

By default, Docker currently uses the `aufs` file system for its back-end storage driver. This module allows the union file system approach of masking important host files and providing private copies of system files to each container. Other options for this driver include `devicemapper`, `zfs`, `btrfs`, and `overlayfs`. The `aufs` driver has been deprecated in the Linux kernel so it is not available upstream which disqualifies it from any forward-looking uses. The `devicemapper` driver, while ubiquitous, shows significant performance degradation and usability issues disqualifying it as well. Both `zfs` and `btrfs` have stability issues and are not yet standard file system drivers so they are also disqualified. The remaining choice is `overlayfs` which has been recently upstreamed into the 3.18 kernel and is the choice for Docker storage driver going forward until `zfs` and `btrfs` become standard [58], [59]. Further limitations, while not as important as the choice of the 3.18 kernel include the availability of `preempt-rt` patches for the kernel.

3.1.3.1 Kernel Configuration

One of the steps involved in tuning the realtime version of the kernel is to apply a specific set of kernel configuration choices during the kernel build process. The kernel shipped with Ubuntu 14.04.3 is version 3.19.0-25-generic. That kernel's default configuration (`.config` file) is used as the basis for the configuration and building of the test kernel version 3.18.20. This default kernel configuration omits even simple optimizations such as disabling CPU frequency scaling, and enabling a preemptable kernel, but the default kernel configuration is likely the most common on servers that have not done significant performance tuning and optimizations. The "performance" kernel used is the 3.18.20-rt18 kernel which is the 3.18.20 kernel with the 3.18.20-rt18 patches, posted 2015-08-11, from [60] applied. The initial kernel configuration used is that of the 3.18.20 kernel. Only a few important parameters of the kernel are modified to tune the systems for performance with the preempt-rt patch. Kernel CONFIG modifications are listed in Table 3.1.

3.1.3.2 Kernel Boot Parameters

Another important mechanism for tuning a kernel is to modify the parameters passed to the kernel at boot time [61]. Standard kernels usually boot with only a few parameters passed to the kernel such as "`ro quiet`" which makes the kernel read-only and suppress most kernel boot messages. The preempt-rt kernel used, Linux 3.18.20-rt18, is booted with a specific set of additional parameters to improve its performance and determinism for guest virtual machines and containers. The set of parameters passed to the performance kernel are summarized in Table 3.2.

One of the most important tuning parameters in a virtual machine host relates to the level of over-subscription for each physical CPU (pCPU). Systems with low

Table 3.1: Kernel Configuration Parameters and Their Descriptions

Parameter	Setting	Description
IKCONFIG	y	Enable config stored in kernel
IKCONFIG_PROC	y	Mount .config as /proc/config.gz
NO_HZ	y	Old idle dynticks config
NO_HZ_FULL	y	Full dynticks system
HIGH_RES_TIMERS	y	Enable high resolution timers
MCORE2	y	Intel Core2 and newer Xeon CPUs features
PREEMPT	y	Fully preemptible kernel
PREEMPT_RT_FULL	y	preempt-rt patch, fully preemptible kernel
HZ_100	y	100HZ kernel clock
HZ_250	n	250 HZ clock turned off
ACPI_DOCK	n	Disable ACPI dock management
ACPI_PROCESSOR	n	Disable CPU power management
CPU_FREQ	n	Disable CPU frequency scaling
CPU_IDLE	n	Prevent linux from controlling CPU idling
BRIDGE	y	Enable Linux Ethernet bridging
INTEL_IOMMU	y	Enable Intel IOMMU
IOMMU_DEFAULT_ON	y	Enable DMA device at boot time
IRQ_REMAP	y	Support interrupt remapping
KVM	y	Supports hosting full virtualization guests
KVM_INTEL	y	Support for KVM on Intel processors with VT extensions
RCU_NOCB_CPU	y	Offload RCU callback processing from boot-selected CPUs

Table 3.2: Kernel Boot Parameters and Their Descriptions

Parameter	Description
`isolcpus=1-4`	Isolate CPUs 1-4, removing them from the scheduler pool of available CPUs.
`hugepagesz=2M`	Set hugepage size to 2 MB (from default 4 KB).
`hugepages=4096`	Allocate 4096 hugepages for a total of 8192 MB.
`nohz_full=1-4`	Set scheduler timer interrupt interval to 1 Hz instead of the 250 Hz default.
`rcu_nocbs=1-4`	Set the specified list of CPUs to be no-callback CPUs. This reduces OS jitter on the offloaded CPUs, which can be useful for HPC and realtime workloads. It can also improve energy efficiency for asymmetric multiprocessors.
`rcu_nocb_poll=1`	Rather than requiring that offloaded CPUs (specified by `rcu_nocbs=` above) explicitly awaken the corresponding `rcuoN` kthreads, make these kernel threads poll for callbacks. This improves the realtime response for the offloaded CPUs by relieving them of the need to wake up the corresponding kernel thread, but degrades energy efficiency by requiring that the kernel threads periodically wake up to do the polling.

performance requirements may stack multiple guest virtual CPUs (vCPUs) on each pCPU. This stacking can lead to multiple vCPUs from unrelated virtual machines sharing the pCPU or host processes running on the pCPU assigned to the guest. Either of these scenarios can have a large impact on guest latency. These resource conflicts can be all but eliminated by "unplugging" pCPUs assigned to a guest from the host scheduler with the `isolcpus` kernel boot parameter. Assigning a pCPU to the isolcpus list, such as `isolcpus=2`, removes that pCPU from that kernel scheduler's CPU pool, preventing the kernel scheduler from running any processes there. The isolated CPUs can still be used to run user processes with Linux utilities such as `taskset` or `numactl`, but remain quiescent until that happens. Scalability of a system with isolated CPUs is reduced, but in an era of server processors with more and more cores, the impact of dedicating individual pCPUs to latency-sensitive processes is also reduced.

Access to memory can be one of the greatest sources of latency in computationally intensive processes. Virtual machines have a similar sensitivity to latency in that memory accesses can require both guest and host-level page walks. In order to minimize the frequency of the multi-level memory lookups, hugepage memory is used as the basis for the realtime guest's memory. As discussed in Section 2.1.3, hugepages can greatly reduce the cost of memory lookups and significantly impact the performance of virtual machines. The host system enabled hugepages in the kernel with the kernel arguments `hugepagesz=2M hugepages=4096`, which allocated 4096 hugepages of 2 MB each.

3.1.3.3 Building a Virtual Machine

The image used for virtual machine testing is based on a raw 16 GB disk image created with the command line tool `qemu-img`. Ubuntu server 14.04.3 is then installed to the image from the ISO file and only the `OpenSSH` package is selected during the in-

Table 3.3: QEMU Virtual Machine Launch Arguments

Parameter	Description
-realtime mlock=on	Lock guest memory pages to prevent them from swapping out.
-mem-path /mnt/huge	Use the specified path for guest memory. Allows utilizing hugepages for guest.
-mem-prealloc	Preallocates guest memory, decreasing latency for most memory accesses.
CPU pinning	Not a QEMU parameter, but use taskset to assign the smp_affinity of each vCPU thread to one of the pCPUs assigned to it.

stallation. After Ubuntu is installed, the packages for `vim`, `ethtool`, `screen`, `qemu-kvm`, `exuberant-ctags`, `apparmor`, `bridge-utils`, and `libpcap-dev` are installed in the VM to keep it consistent with the host installation.

3.1.3.4 QEMU and Hypervisor Parameters

QEMU is responsible for device emulation so selection of its arguments can have a significant impact on VM performance. QEMU is used to launch the guest virtual machine with some additional command-line arguments to improve guest and hypervisor performance. The complete set of additional QEMU parameters is shown in Table 3.3. In this work, higher-level libraries such as `libvirt` or VMware orchestration are not used in favor of running as efficiently as possibly without additional software layers of abstraction impacting performance. The hope is for maximal performance so minimal abstraction layers are utilized in the system configuration.

3.1.3.5 Docker Parameters

The Docker environment is logically separated into building with `docker build`, back-end management performed by the `docker daemon`, and the `docker run` command that actually instantiates containers from images and assigns the appropriate `cgroups` and `namespaces` to the containers. The `docker daemon` sets up the software environment for the containers, specifically their file systems, libraries, and available binaries. Along with the files available to the container, the daemon also manages `cgroups` and `namespaces` to properly control the container's resources and limit its scope. The `docker run` command is responsible for requesting resources and privileges for the container as it is instantiated.

The image building process of Docker is controlled by the `docker build` command line tool. This tool takes recipe files known as Dockerfiles [62] and constructs the image based on instructions therein. It is useful to think of the relationship between an image and a container similarly to the program/process dichotomy. The image is simply a container's data and the container is a running image. A Dockerfile specifies the base image that is used for the build along with setting environment variables and installing necessary packages and binaries to the container's file system. It is read by the `docker build` tool and the image is constructed as the composite overlay of each step in the build process [63]. All of the containers used are based on the official Ubuntu 14.04 LTS image. The container base image is chosen in order to maintain a consistent environment with the host and virtual machines. The image created for benchmarking pulled the Ubuntu base image, applied labels and environment variables, then installed `netperf`.

The daemon has a number of settings that may be modified to tune parameters such as available cgroup CPU sets, bridged networking, and the storage driver used

for container file systems. In this study, the `docker daemon` is launched with standard parameters but for the following exceptions. As discussed in Section 3.1.3, the `overlayfs` storage driver is chosen, requiring the `docker daemon` to be launched with `--storage-driver=overlay`. Additionally, Docker's default bridge, `docker0`, is configured to draw on a pool of IP addresses from the subnet 192.168.42.240/28 for its bridged virtual Ethernet interfaces. The additional arguments used with `docker run` are only used to assign the cgroup cpusets and corresponding memory node for the realtime kernel. These arguments comprised `--cpuset-cpus=1-4` which tells Docker to create a cgroup containing those CPUs and assign the container to that cgroup and `--cpuset-mems=0` which tells Docker to assign socket 0 (the local socket) as the only available memory node.

All containers are run with non-persistent file systems by using the `--rm` option in the `docker run` line. This had the effect of removing the container's file system overlay after it exits, preventing any file system modifications between runs. This is one of the advantages of containers in obtaining consistent performance since the container file system is reloaded from the base image at every launch. Containers are also run with the flags `-i -t` or `-it`, which requests that an interactive `tty` be allocated to the container while it is running. This incurs a small additional overhead for the terminal process, but this is consistent with the default operation of Docker. Containers are all launched with a specified MAC address for their bridge interface to remove the need to refresh address resolution protocol (ARP) tables with new, randomized MAC addresses.

3.2 Experimental Procedure

This study compares the performance of two types of virtual systems with tuning to those without and compares the performance of two different network interfaces.

The primary experimental variable is the selection of kernel with or without tuning, represented by the 3.18.20 and 3.18.20-rt18 kernels, respectively. The second important option in the experiment is the choice of network interface comprising bridged or physical and their effect on performance. KVM virtual machines and Docker containers represent the options for choice of virtual system type studied. The set of 3 variables with 2 parameters each produced 8 system configurations to test.

The performance of latency-sensitive network workloads are often not bound by memory or CPU, but by their ability to process I/O traffic. Testing herein concentrated on network-related benchmarks to evaluate bandwidth and latency of the various operating system and network configurations. The network paths that are tested comprised both TCP and UDP flows in addition to ICMP latency.

Network performance is measured with `netperf` [64] and `ping`. Using `netperf` version 2.7.0 [65], [66], network bandwidth and latency are measured with tests including TCP_STREAM, TCP_RR, UDP_STREAM, and UDP_RR. For each of the `netperf` tests, 20 replicate samples of ten seconds each are collected in series. For the `ping` test, 100 samples are collected with a one second interval.

In most cases, the STREAM bandwidth tests of `netperf` rely on the kernel for ARP and routing. When attempting to measure UDP streaming performance over bridged interfaces to the container with default settings, the client system was not able to establish a connection, displaying an error that indicated a routing problem. The cause of this error is a setting in the `netperf` UDP STREAM test that disables IP routing for that test by default [66], [67]. On inspection of the `netperf` source code, it was found that this setting was "grudgingly added" to prevent accidentally flooding default routes and corporate networks with UDP STREAM packets [66]. The consequences of this choice are only apparent when routing to a different subnet is required, and the client process is unable to route its packets to the server. For each

of the network tests performed, both bridged and physical interfaces are assigned to different subnets in each virtual system to expedite testing. Therefore, in order to maintain consistency between interfaces, all UDP STREAM tests enable routing with the `netperf` flag `-R 1`.

Additionally, the UDP STREAM tests show unexpected behavior when the message size used in the test is left at its default size. The default message size of 64 kb is fragmented into smaller data packets to fit within the default Ethernet frame size of 1500 bytes. If one of the fragmented UDP packets is lost in transit, re-assembling the message fails and that message is counted as lost in bandwidth calculations. Although the network connections between the test systems did not carry any additional traffic, this type of packet loss under the tested conditions has been difficult to predict. To avoid the issue of unpredictable packet losses having a significant impact on bandwidth calculations, the UDP STREAM tests are all run with a non-standard message size of 1472 bytes, using the flag `-m 1472`.

For each of the request-response (RR) tests performed, the size of the messages exchanged may be set at runtime with the test-specific command line parameter `-r 1,1`, which sets the payload size as one byte for each direction. A small 1-byte payload was used in all of the RR tests to minimize the potential for additional variance due to handling larger packets.

Chapter 4

RESULTS

Network performance of Docker containers and KVM/QEMU virtual machines is analyzed using standard benchmarking tools including `netperf` and `ping`. The results of the tests are presented as box plots with the median in the center, 25th and 75th percentiles at the bottom and top bounds of the box and minimum and maximum at the ends of the error bars. This plot helps to illustrate the one-sided distribution that may often be observed in network performance analysis where measurements often approach the theoretical maxima for each test, but may have long tails in the other direction.

4.1 Network Bandwidth

Network bandwidth is measured for all test configurations with both TCP and UDP streaming flows. It should be noted, however, that the maximum theoretical speed for Ethernet connections, both optical and copper, is just below 95% of the physical data rate. This is due to the overhead of default maximum transmission unit (MTU) Ethernet frames (1518 bytes + 8 byte preamble and 12 byte (96 ns) interframe gap (IFG)), Ethernet header (14 bytes), 4 byte Frame Check Sequence (FCS), 20 byte IP header, 20 byte TCP header, and 1460 MSS for TCP. Overall, this amounts to 78 bytes of overhead at various layers to transmit 1460 bytes of data representing 94.9% efficiency in the best case. Thus, any performance above 9.4 Gbps (9400 Mbps) is considered line rate. The overhead fraction can be reduced with jumbo Ethernet frames, but that level of network tuning is outside the scope of this study.

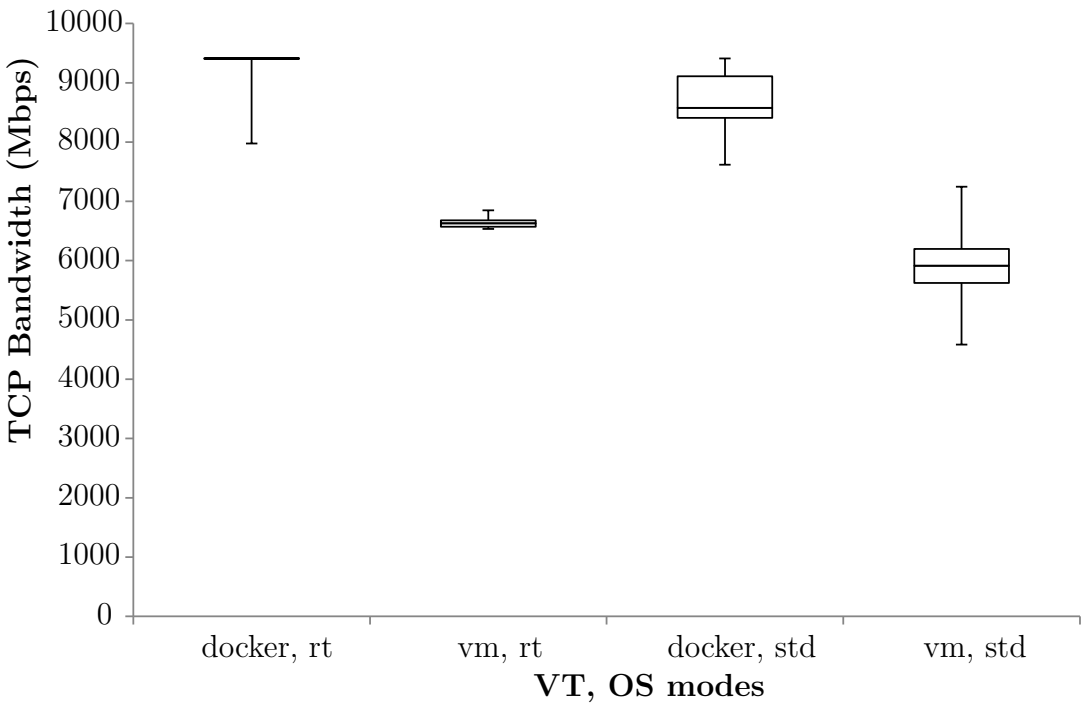

Figure 4.1: TCP Streaming Bandwidth: Bridged Network

4.1.1 TCP Bandwidth

Bandwidth of the various network interfaces and kernels was measured using the TCP STREAM bandwidth test in `netperf`. The results of the bridged measurements are shown in Figure 4.1. It is evident in this figure that the streaming TCP bandwidth of both environments improves with tuning as well as decreases in variance. It is impressive that the realtime container reached line rate with only a few points below the median. The performance of both the container and virtual machine are consistent with a realtime kernel in that the performance is essentially deterministic with low variance. The standard deviation, while not the best statistic for displaying the distribution of these data, is the typical measurement of variation. Standard deviation of the virtual machine in this case decreased by 87% and decreased by 4.4% for the container. This is the result that is desired when tuning a system for deterministic performance.

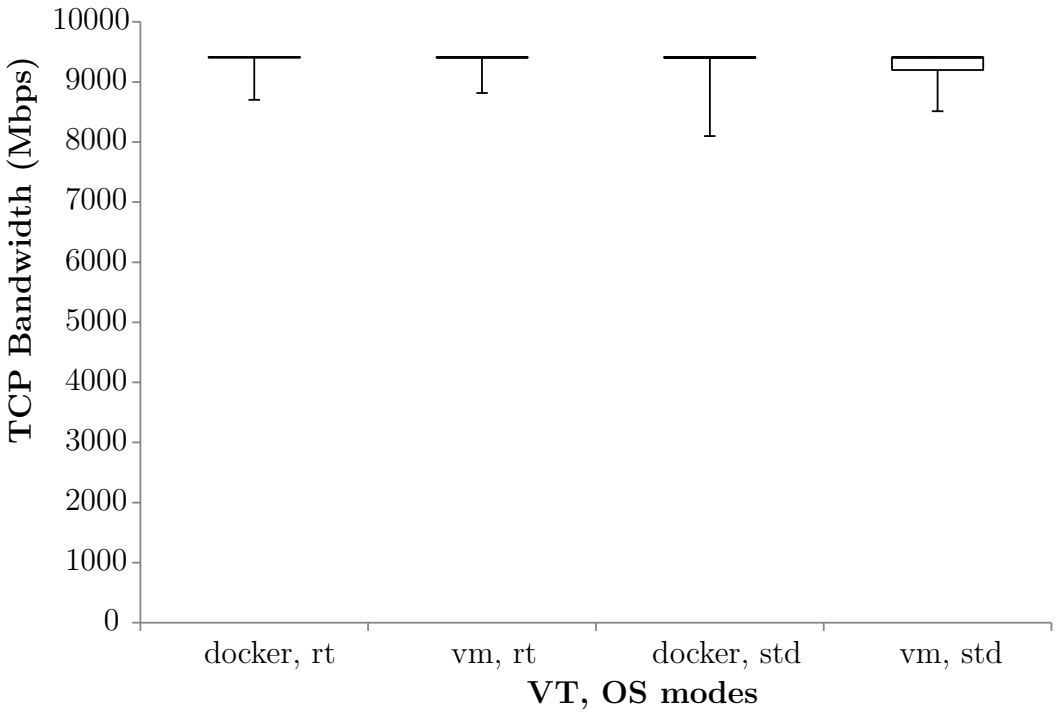

Figure 4.2: TCP Streaming Bandwidth: Physical Network

The results of the TCP STREAM test on the physical connection are shown in Figure 4.2. This figure shows high throughput from the virtual environments with all of the variants reaching line rate. While line rate performance was somewhat expected in the physical interfaces, even the standard kernel variants show exceptional performance. The realtime tuned kernel decreases variance in this case, but, since the baseline performance is so high already, little to no gain is observed in the median performance, with all configurations less than 0.01% separated from each other. The differences between the bridged and physical network performance seen in Figures 4.1 and 4.2, respectively, partly illustrate the risks of using host bridging for networking due to the additional latency contributed by a higher density of guest systems on that host.

4.1.2 UDP Bandwidth

Although a large fraction of data center traffic has been shown to use TCP [49], one of the greatest sources of traffic in the Internet is streaming video and audio which are typically UDP flows. UDP uses less computation per byte of goodput since flow control and loss correction are not part of the protocol. This seems to indicate that UDP performance should be equivalent to or higher than TCP performance. The UDP bandwidth results observed, however, are contrary to that expectation, due in part to the protocols used in the UDP STREAM tests.

Figure 4.3 shows the results of UDP bandwidth measurements over bridged network connections. As described in Section 3.2, however, the UDP STREAM test has some important details that should be understood before interpreting its performance. Unexpectedly, one important caveat is that `netperf` disables IP routing for the UDP STREAM tests by default. IP routing is enabled in all UDP STREAM tests to facilitate routing across separate subnets for each interface. Another consideration with the `netperf` UDP STREAM tests is the selection of the message size transmitted at each iteration. The default message size for this test is 64 KB, but a message of this size is fragmented as it traverses an Ethernet segment with a 1500 byte MTU. The `netperf` UDP bandwidth test counts the entire message as lost if a single fragment is dropped in the network. Peak bandwidth on this test is observed to be higher with a larger message size, but the results of that test configuration are difficult to reproduce. To prevent this error condition and ensure consistent measurements, the message size is limited to 1472 bytes for all UDP STREAM tests.

Low UDP bandwidth is a direct result of limiting the message size as described above. Containers showed minimal variance in this test for both kernels, but improved throughput for the realtime kernel. This indicates that the computational cost of

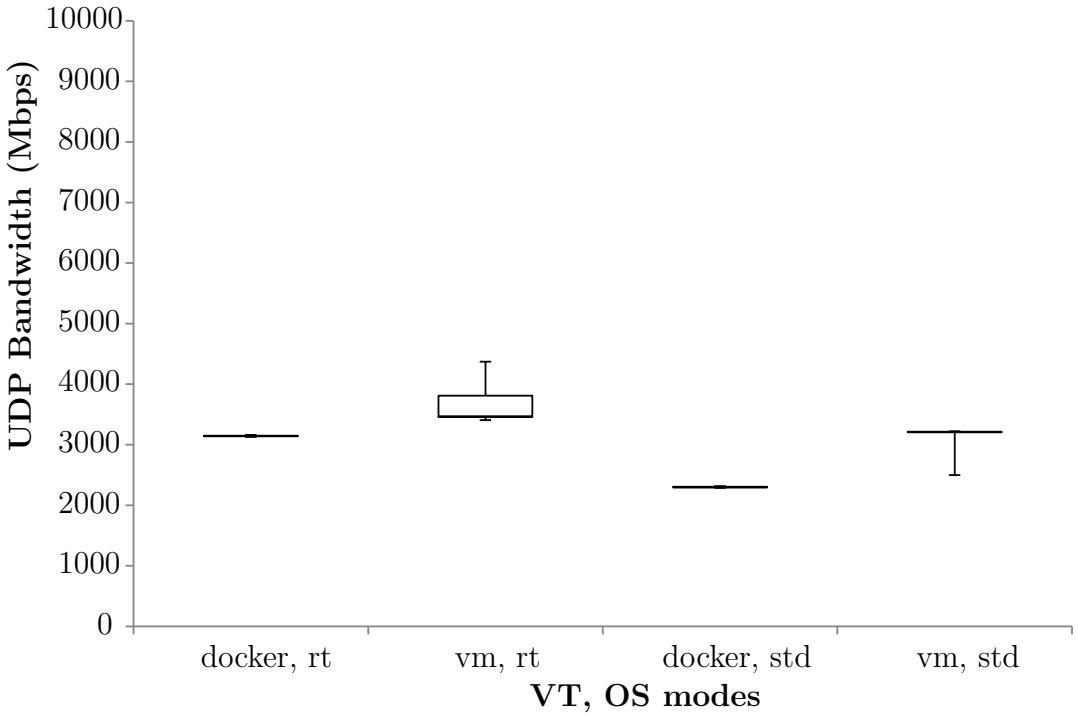

Figure 4.3: UDP Streaming Bandwidth: Bridged Network

determining bridge actions in the kernel is not a significant source of latency at this message size and resulting data rate. Although the VM showed an increased bandwidth in the realtime kernel, the standard deviation of observed measurements also increases from 722 to 964 Mbps, which is contrary to the goals of system tuning.

This first set of UDP test results is indicative of the complications encountered in this thesis during UDP testing with netperf. It is known that UDP is an unreliable message delivery protocol so it seems incongruous to send large messages that are not counted when fragmented over UDP. Without the large messages, however, `netperf` UDP STREAM has limited performance. This presents an interesting contradiction in the UDP operation of `netperf` and the use of UDP in the network.

UDP bandwidth is also measured over a physical network interface and its results are shown in Figure 4.4 . These results again show how significantly bandwidth is limited by the restriction of the UDP message size. Although slightly higher performing

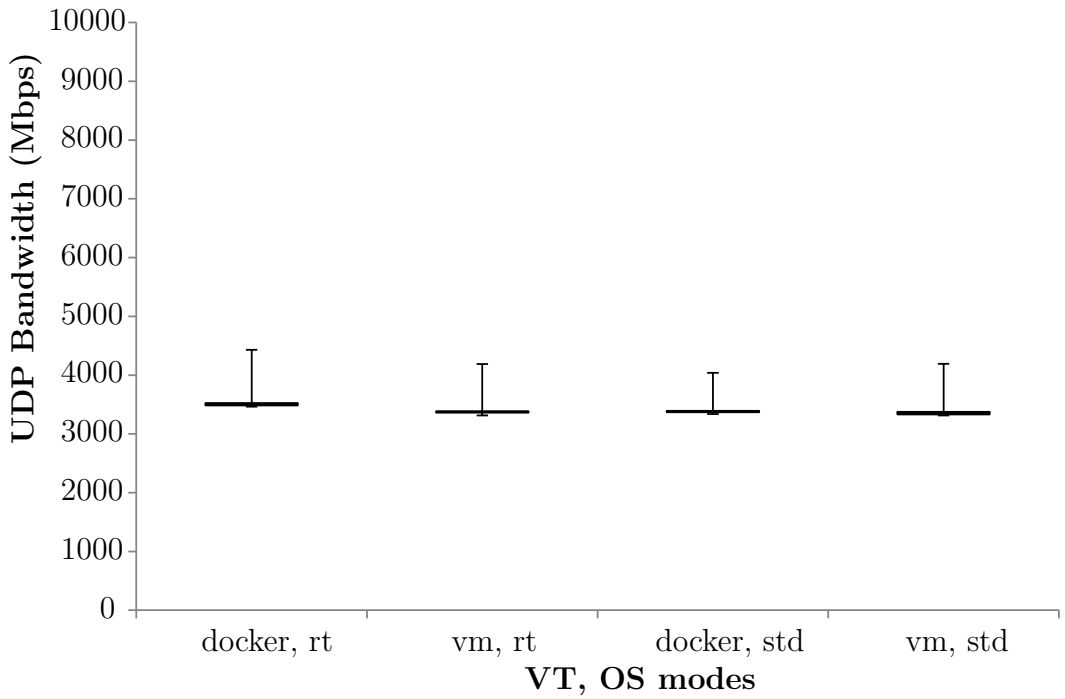

Figure 4.4: UDP Streaming Bandwidth: Physical Network

than the bridged interfaces, the maximum bandwidth observed, 4.43 Gbps in the realtime Docker test, is only 1.5% higher than the maximum bandwidth observed in the bridged interface, 4.37 Gbps for the realtime virtual machine. None of the UDP bandwidth results approach the theoretical limit of 9.5 Gbps nor do they vary significantly between test conditions. Since all four systems show similar maxima and variation, it appears that the system load created by this test is insufficient to challenge system performance. Given the lack of variation among any of the test systems observed, it is difficult to draw any conclusions other than that the workload did not provide a significant challenge to the systems. The potential for improving these conditions is discussed further in Section 5.1.1.

4.2 Network Latency

Bandwidth measurements get a lot of attention due to the desire to push more packets through the high-bandwidth connections of data centers and network backbones. Latency and jitter however, are often the critical metrics when evaluating a network workload because low bandwidth does not prevent streaming services from running effectively whereas high latency can cause dropped calls or prevent connectivity entirely. In this section, the results of `netperf` and `ping` latency measurements are discussed.

Variation in the measured latency is partly due to non-determinism normally present in operating systems, but a small component of the total latency is due only to the network delays. The 10 Gbps optical cables connecting the two systems are approximately 2 meters long so the propagation delay along these connections should be about 6.7 ns. In-network queuing delay can be ignored since the systems are directly connected and queuing is only occurring at their network interfaces. The packetization delay for transmitting or receiving a maximum-size Ethernet frame of 1500 bytes is 1.214 μs. This represents as much as 3% of the total latency observed, but this offset should be constant across all tests and have little effect on the observed variance.

The `netperf` TCP Request/Response (RR) subtest and UDP RR subtest both measure the number pf transactions that can be completed for a given request and response size during a specified time period [69]. Each transaction comprises the exchange of a single request and a single response so the average round trip latency can be determined from the transaction rate. The time required to initiate and then tear down a network connection is not included with this test. Only a few packets are sent with this test so congestion is not a concern in this environment, but CPU involvement in the packet processing may comprise a significant portion of the latency.

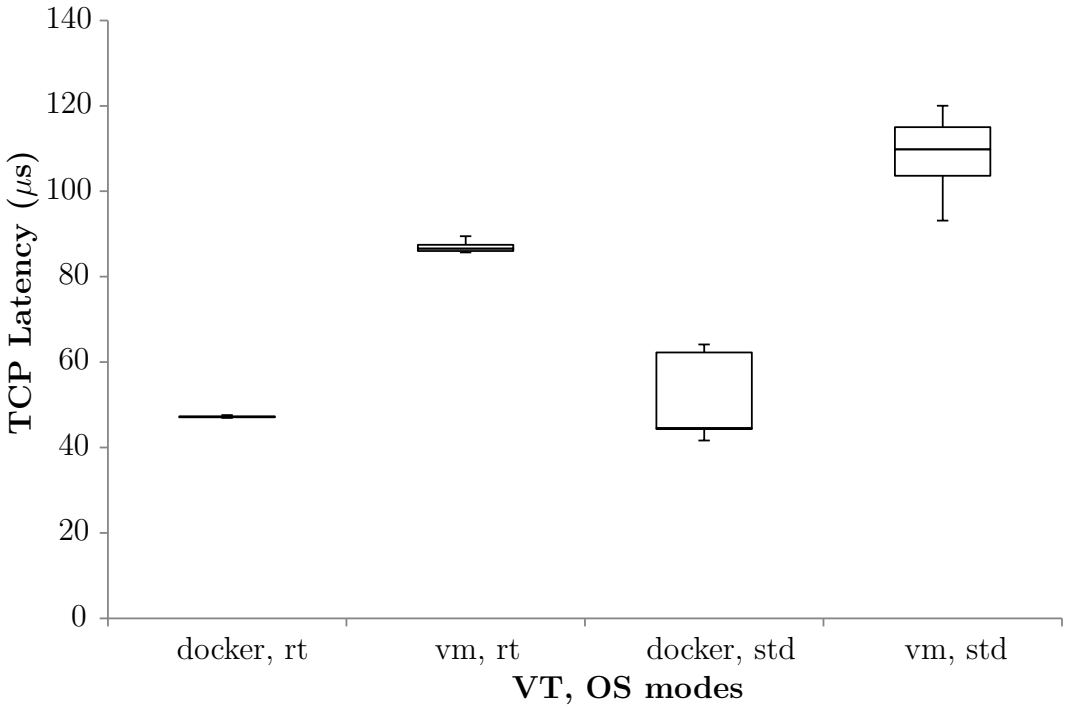

Figure 4.5: TCP Request/Response Latency: Bridged Network

To minimize bias and variance added to measured latencies, the request-response test variants are all performed with 1-byte payloads for request and response packets.

4.2.1 TCP Latency

Figure 4.5 shows the TCP RR latency of the bridged interfaces. This figure illustrates the intended result for tuning a system toward realtime performance. Both the virtual machine and container show a significant reduction in variance when using a realtime kernel instead of a standard kernel. The realtime Docker result does not show much decrease in its median latency, but variance decreases dramatically as expected. The virtual machine, however, showed reductions in latency as well as its variance.

Measurements of TCP RR latency over a physical interface are shown in Figure 4.6. Similar to the bridged TCP STREAM bandwidth, the physical interface shows little improvement moving from the standard to a realtime kernel, but all four modes seem

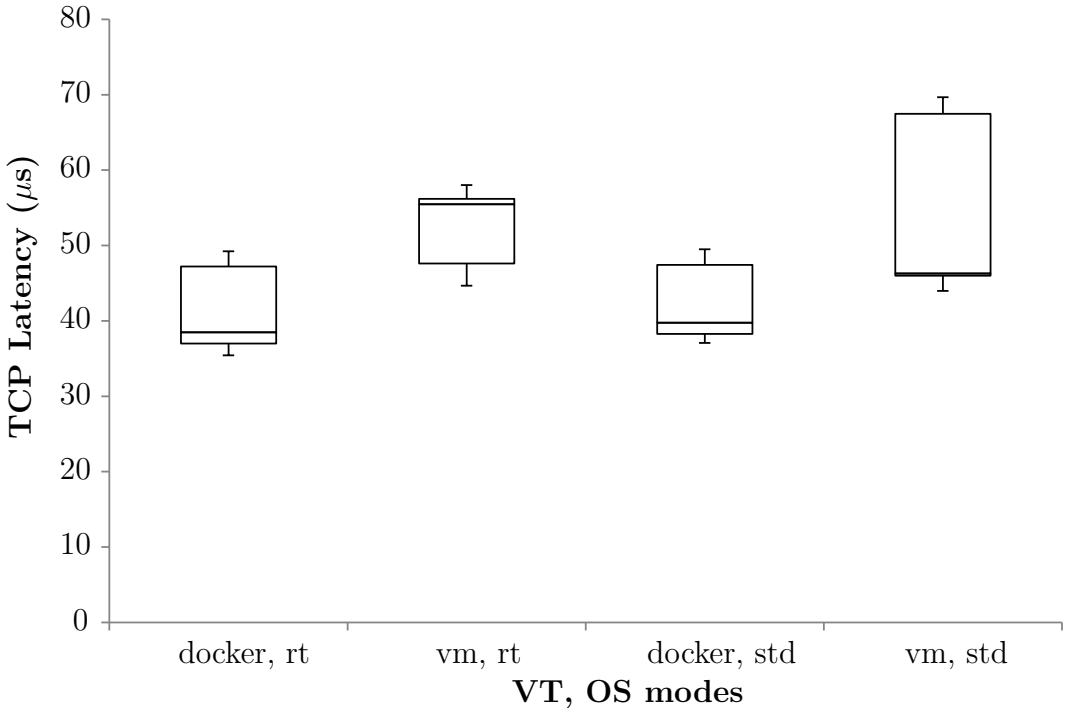

Figure 4.6: TCP Request/Response Latency: Physical Network

to perform similarly. This trend represents one of the intended benefits of physical interfaces where performance is expected to be more consistent across interfaces and virtual systems due to reduced kernel involvement. Variance within each environment is not especially low, but the physical interface is much more consistent between systems than the bridged interface, due to fewer abstraction layers involved in processing packets on this interface. What is surprising, however, is that the median latency of the virtual machine shows an increase for the realtime kernel, but a decrease in its variance. This may be normal with a realtime kernel, however, where improved determinism may come at the cost of maximum performance. It also seems unusual that the Docker measurements do not seem to vary. On closer inspection, they are close, differing by less than 5% at their minima and less than 1% at their maxima.

4.2.2 UDP Latency

Although the TCP measurements presented so far are consistent with expectations, the UDP workloads have not been as straightforward. The latency of both of the UDP Request-Response tests are an example of the unexpected behavior observed. When measuring network latency, it is expected that the majority of the delay comes from transit across the network and queuing delays at intermediate nodes along the way. The UDP latency shown in Figure 4.7, however, shows that the network effects discussed previously can be small in comparison to other effects in the system. These UDP RR latency measurements are generally higher than the bridged TCP measurements, but the latency measured with the standard virtual machine configuration is actually lower than the TCP case. These measurements were repeated multiple times with similar results each time, indicating that there is a common configuration limiting the rate at which the UDP test can run.

Results of the UDP Request-Response measurements over a physical interface are shown in Figure 4.8. This figure shows the same relatively high latency as observed over the bridged connection with unexpectedly low variation. The Docker results are also surprising due to the increase in variance observed in the realtime tuned system over the standard Docker system. The virtual machine results are suspect because both measurements are very precise at 125 ± 0.006 μs. The combination of higher latency in the physical measurements and the very high precision observed in the virtual machine support the idea that the rate of UDP packet transmission is being kept artificially low in this case. No settings in the Linux network stack could be found to modify this transmission rate. Investigation into the source code of `netperf` did not uncover any special treatment of the UDP RR tests leading to their unexpected

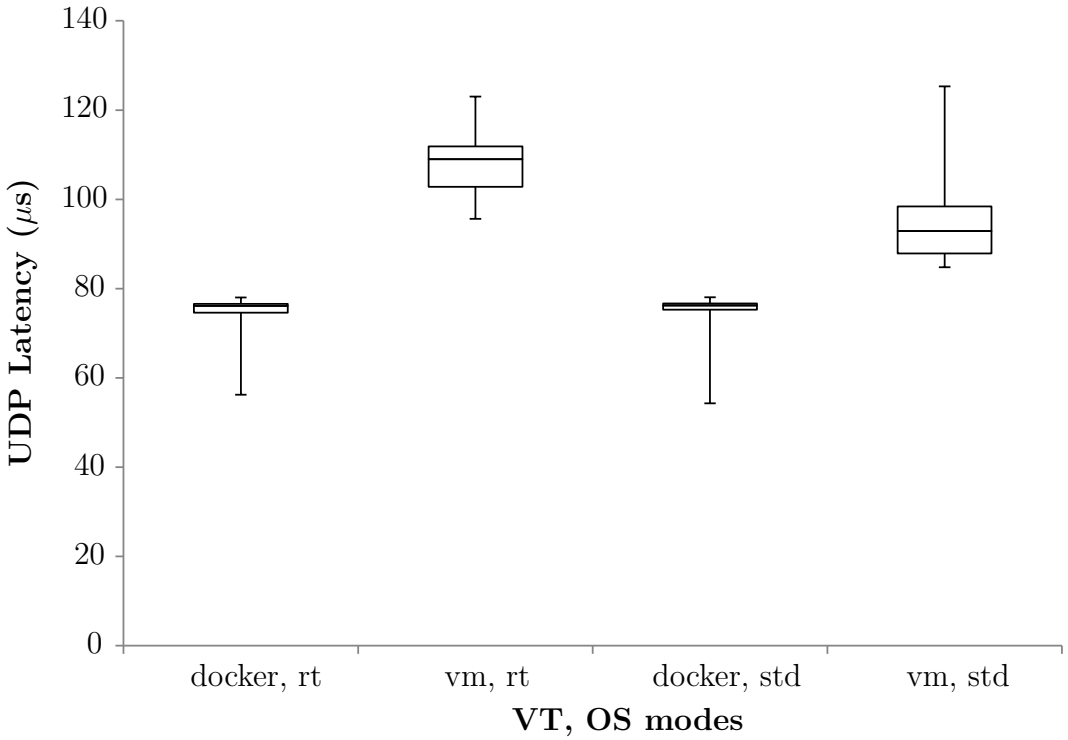

Figure 4.7: UDP Request/Response Latency: Bridged Network

behavior, but the unusual treatment of UDP traffic in the STREAM test does not add confidence to `netperf` handling of UDP latency measurements.

4.2.3 ICMP Latency

In order to get an idea of how the whole system responds to a realtime kernel, ICMP latency was also measured with the standard network tool `ping`. The `ping` tool presents a slightly different case than `netperf` due to the location of the responding agent. Since `netperf` is a userspace process, its CPU affinity may be set to any CPU in the system, specifically those that have been isolated from the kernel. ICMP responses, however, originate from the kernel's network stack so cannot be easily assigned to a CPU. This limits how isolated its performance can be from the rest of the system. The results of the `ping` testing of the bridged connections are shown in Figure 4.9. It

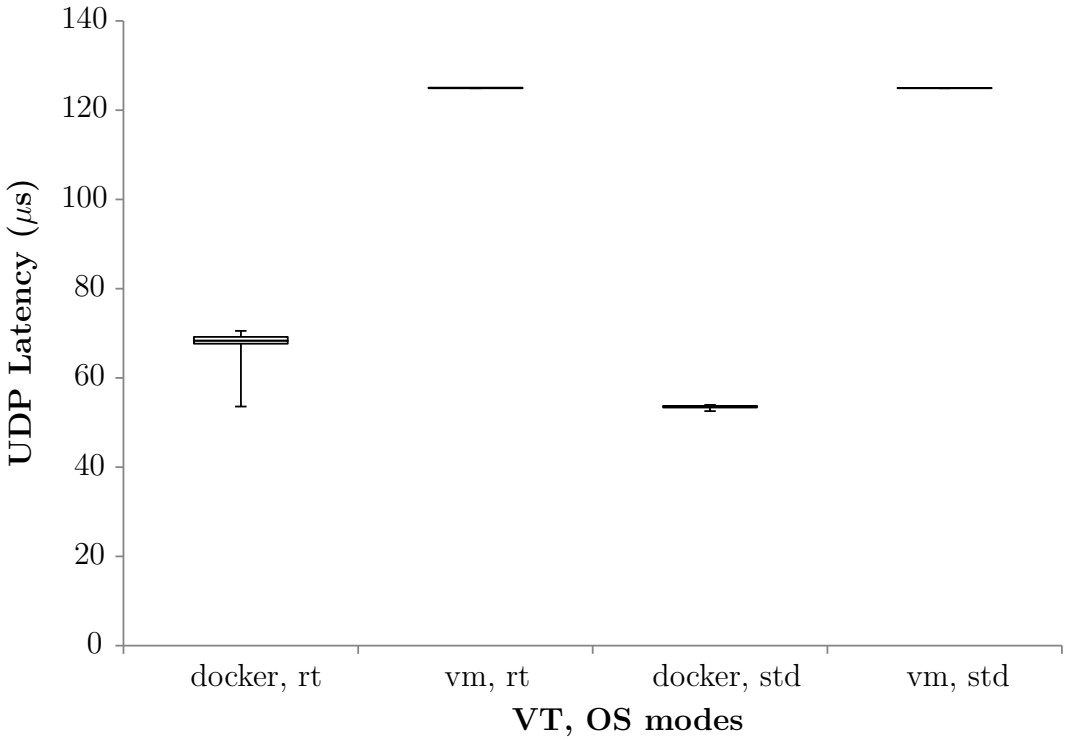

Figure 4.8: UDP Request/Response Latency: Physical Network

is interesting that the container performance improved only slightly with the realtime kernel, whereas the virtual machine variance improved significantly at the cost of higher median latency. When using the standard kernel, the virtual machine sees very high peak latency due to the additional abstraction layers that must be traversed for each packet and the nondeterministic floating CPU cores that might respond. The difference between the systems can be explained with an understanding of the CPU affinity of the virtual machine. When the VM is pinned to specific, isolated CPU cores as in the realtime example, the host kernel is not involved in the ICMP response so it does not contribute to the variance. The containers, however, do not have this same benefit. Although processes running inside a container are pinned to the container's cgroup cpuset, the host kernel is responsible for responding to ICMP requests. Since

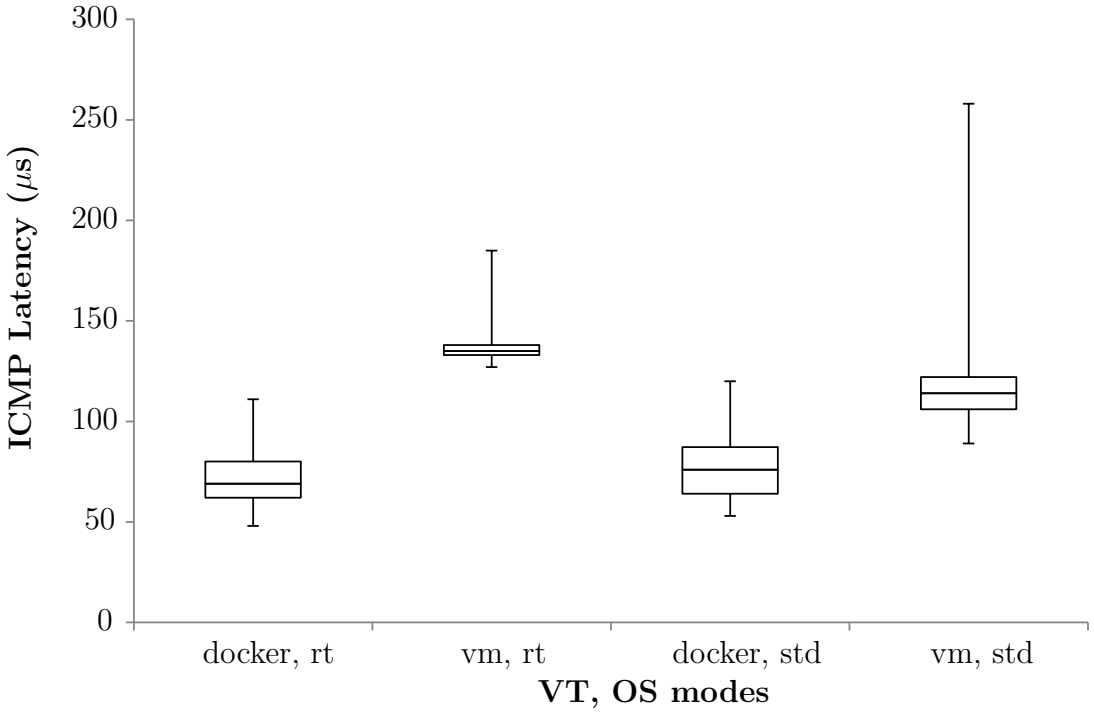

Figure 4.9: ICMP Latency: Bridged Network

the responding agent in the host kernel removes much of the benefit of isolating cores, both kernels should behave similarly in this test.

Aside from the expected decrease in latency from the bridged interfaces to the physical, the results of the physical interface `ping` testing are very similar to those of the bridged interfaces, Figure 4.10. This is a result of the constraints mentioned in the discussion of the bridged ICMP results. The host kernel latency is only slightly affected by kernel tuning, but the guest kernel gets private cores to run its processes, allowing a modest decrease in both variance and magnitude. Both of these ICMP latency tests serve to illustrate that `ping` is only marginally helpful at assessing the effect of kernel tuning on guest system performance.

In the preceding sections, the variance of TCP connections is shown to improve with a realtime kernel and tuning to enable process isolation. Additionally, physical interfaces are shown to outperform bridged interfaces in almost all cases, demonstrat-

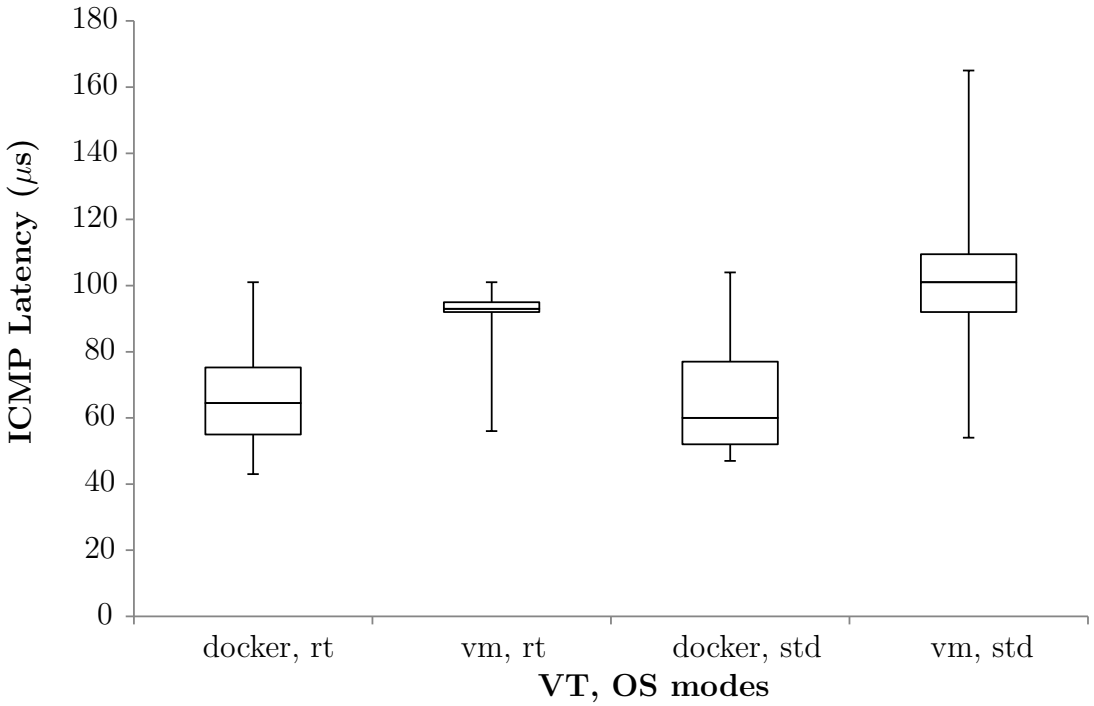

Figure 4.10: ICMP Latency: Physical Network

ing that bridged interfaces can create significant bottlenecks to performance. UDP measurements shown are not as illuminating, however. In each UDP test performed, there exists some system configuration or program setting that complicates interpretation of the results. Modification of the message size for the UDP STREAM tests was required in order to obtain consistent results for that test. That had the unintended effect of diminishing the value of those results in understanding the effects of system tuning. Finally, the results of the ICMP testing serve to demonstrate that tuned kernels and system configurations can improve performance in most cases, but the environment and context where a process is running are equally important to performance as system configuration.

Chapter 5

CONCLUSIONS & FUTURE WORK

Uses of virtualization have consistently grown as new technologies have improved performance and the overhead of virtualization has decreased. Containers are a recent innovation in virtualization that has been a catalyst for change in the industry as NFV and new computing paradigms are developed. The open-source operating system Linux has seen significant improvement in both traditional virtualization and the creation of new mechanisms for process isolation. This innovation has led to new use cases and orchestration mechanisms to manage and control virtual systems.

As expected, the Docker containers studied generally had higher bandwidth and lower latency than virtual machines. In almost all of the streaming TCP tests both systems realized a reduction to variance in their bandwidth and latency. Virtual machines and containers tended to perform similarly, however, when the physical network interfaces were dedicated to the virtual system. The reduction in variance often comes at the cost of a small decrease in performance, but, as in the case of the TCP STREAM test over a bridged interface, throughput may improve in some cases. Measurements of UDP streams showed unusual behavior, caused by unexpected handling of UDP fragmentation and potential limitations to the flow rate of UDP packets.

Although bridged interfaces saw a greater improvement in most cases than the physical interfaces, it was also shown to have more variability and lower throughput than the physical network connections. This paradigm is a concern for large numbers of guests because any networking topology that requires host kernel involvement cannot scale to a large number of guests without a performance impact. This is because the

work required by the host to monitor and handle bridge traffic and layer 3 decisions for multiple guests can be significant and induces additional latency for each guest added to the bridge. Bridged interfaces are extremely flexible and versatile, but the computation required by the host kernel indicates that a large number of client systems would see significant contention for the bridge, limiting its potential for scale out.

Specific innovations such as the preempt-rt patches for Linux have enabled levels of determinism that were previously only available in embedded systems with dedicated hardware. Application of the `preempt-rt` patches and targeted tuning of the host and guest systems was shown to enable some increases in TCP bandwidth as well as reduction of variance in TCP request/response latency, especially in network interfaces such as bridges which require host kernel involvement. While no work was performed to quantify their contributions, isolating processes from the OS general scheduler seems to yield a significant improvement to their performance. This trend of dedicating hardware to individual processes should only increase as the core counts of server systems increase. A system running a processor with many CPU cores, such as the 60-core Xeon Phi, could be the server core of the future in this context.

5.1 Future Work

5.1.1 UDP Flows

Chapters 3 and 4 described unexpected UDP performance and required workarounds when UDP flows were measured with `netperf`. It seemed that `netperf` was configured with settings that prevented the UDP flows from transmitting at the expected rate, but the root cause of this behavior is still unknown. Since `netperf` is one of the most common network performance tools, the solution to the unexpected behavior could provide valuable insight into network performance analysis. Benchmarking utilities

such as iperf3 [68] have been addressed as alternatives to `netperf` and are perhaps better suited to UDP measurement.

5.1.2 Traffic Analysis

Transport protocols such as TCP and UDP comprise a large fraction of the traffic in datacenters and the Internet. Although this study examined network performance of TCP and UDP flows in particular, there exist complex relationships between the real-world workflows and the resulting subsystems that are stressed in their hosts. Further investigation should be made into the impact of simultaneous network flows that represent a more realistic balance of traffic through network backbones. These studies could help to illuminate weaknesses in the network stack and related subsystems and upstream those findings into the `preempt-rt` patch set and network drivers.

5.1.3 Virtual Functions

Virtual functions enabled by SR-IOV are a mechanism by which physical devices may be multiplexed and share their resources among many virtual devices. While virtual functions may subdivide the resources of the physical device, a virtual function belonging to a peripheral such as a network interface card operates at line rate and should attain latency similar to the physical function. Although scaling to tens of containers may not be possible with physical device assignment, a small set of physical devices may be subdivided into virtual functions, enabling near-line-rate performance for all of the containers. The performance of virtual functions should be investigated in the context of process isolation and system tuning to verify that containers can scale out to the levels advertised in the press.

5.1.4 DPDK

The isolation of processes from the rest of the system has been shown to improve performance. User space libraries such as DPDK can allow systems to minimize interference from the kernel by removing the interfaces from the kernel domain and using poll mode user space drivers to minimize response latency [50]. This paradigm represents the limit of process and hardware isolation because applications written with the DPDK library monopolize the network interfaces and CPU cores, consuming extra hardware resources to maximize performance. Optimizing parameters of the network stack in Linux or implementing benchmarks in DPDK are additional optimizations for improving network performance and should be considered in future work.

REFERENCES

[1] M. Cohn, *We is us - OPNFV & ETSI accelerate NFV adoption*, https://www.sdxcentral.com/articles/contributed/opnfv-etsi-nfv-nfv-adoption-marc-cohn/2015/06/, [Online; accessed 10-October-2015], Jun. 2015.

[2] OPNVF, *Telecom industry and vendors unite to build common open platform to accelerate network functions virtualization*, https://www.opnfv.org/news-faq/press-release/2014/09/telecom-industry-and-vendors-unite-build-common-open-platform, [Online; accessed 19-September-2015], Sep. 2014.

[3] A. Younge, R. Henschel, J. Brown, G. von Laszewski, J. Qiu, and G. Fox, "Analysis of virtualization technologies for high performance computing environments", in *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, Jul. 2011, pp. 9–16. DOI: 10.1109/CLOUD.2011.29.

[4] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. De Rose, "Performance evaluation of container-based virtualization for high performance computing environments", in *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*, IEEE, 2013, pp. 233–240.

[5] B. Des Ligneris, "Virtualization of Linux based computers: The Linux-VServer project", in *High Performance Computing Systems and Applications, 2005. HPCS 2005. 19th International Symposium on*, IEEE, 2005, pp. 340–346.

[6] Linux Foundation, *The Linux Foundation announces project to advance real-time Linux*, http://www.linuxfoundation.org/news-media/announcements/2015/10/linux-foundation-announces-project-advance-real-time-linux, [Online; accessed 5-October-2015], 2015.

[7] J. Edge, *The future of the realtime patch set*, http://lwn.net/Articles/617140/, Online; accessed 2-September-2015, Oct. 2014.

[8] docker.com, *Docker – build, ship and run any app, anywhere*, https://www.docker.com, [Online; accessed 24-October-2015], Oct. 2015.

[9] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors", in *ACM SIGOPS Operating Systems Review*, ACM, vol. 41, 2007, pp. 275–287.

[10] J. N. Matthews, W. Hu, M. Hapuarachchi, T. Deshane, D. Dimatos, G. Hamilton, M. McCabe, and J. Owens, "Quantifying the performance isolation properties of virtualization systems", in *Proceedings of the 2007 workshop on Experimental computer science*, ACM, 2007, p. 6.

[11] M. Gomes Xavier, M. Veiga Neves, F. de Rose, and C. Augusto, "A performance comparison of container-based virtualization systems for mapreduce clusters", in *Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on*, IEEE, 2014, pp. 299–306.

[12] K.-T. Seo, H.-S. Hwang, I.-Y. Moon, O.-Y. Kwon, and B.-J. Kim, "Performance comparison analysis of Linux container and virtual machine for building cloud", vol. 66, 2014, [Online, downloaded 26-September-2015].

[13] R. Morabito, J. Kjällman, and M. Komu, "Hypervisors vs. lightweight virtualization: A performance comparison", in *Cloud Engineering (IC2E), 2015 IEEE International Conference on*, IEEE, 2015, pp. 386–393.

[14] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and Linux containers", in *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*, Mar. 2015, pp. 171–172. DOI: 10.1109/ISPASS.2015.7095802.

[15] M. S. Rathore, M. Hidell, and P. Sjödin, "KVM vs. LXC: Comparing performance and isolation of hardware-assisted virtual routers", *American Journal of Networks and Communications*, vol. 2, no. 4, pp. 88–96, 2013.

[16] M. J. Scheepers, "Virtualization and Containerization of Application Infrastructure: A Comparison", in *21st Twente Student Conference on IT*, Jun. 2014, pp. 1–7. [Online]. Available: http://referaat.cs.utwente.nl/conference/21/paper/7449/virtualization-and-containerization-of-application-infrastructure-a-comparison.pdf.

[17] Z. Wang, X. Zhu, P. Padala, and S. Singhal, "Capacity and performance overhead in dynamic resource allocation to virtual containers", in *Integrated Network Management, 2007. IM '07. 10th IFIP/IEEE International Symposium on*, May 2007, pp. 149–158. DOI: 10.1109/INM.2007.374779.

[18] J. Che, Y. Yu, C. Shi, and W. Lin, "A synthetical performance evaluation of OpenVZ, Xen and KVM", in *Services Computing Conference (APSCC), 2010 IEEE Asia-Pacific*, Dec. 2010, pp. 587–594. DOI: 10.1109/APSCC.2010.83.

[19] L. Fu and R. Schwebel, *RT preempt howto*, https://rt.wiki.kernel.org/index.php/RT_PREEMPT_HOWTO, [online; accessed 22-September-2015], Sep. 2015.

[20] J. M. Welch, *GENI_VT*, https://github.com/matt-welch/GENI_VT, [github repo; accessed 29-October-2015, Oct. 2015.

[21] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures", *Commun. ACM*, vol. 17, no. 7, pp. 412–421, Jul. 1974, ISSN: 0001-0782. DOI: 10.1145/361011.361073.

[22] A. Liguori, *The myth of type I and type II hypervisors*, http://blog.codemonkey.ws/2007/10/myth-of-type-i-and-type-ii-hypervisors.html, [Online; accessed 22-September-2015], 2007.

[23] J. S. Robin and C. E. Irvine, "Analysis of the Intel Pentium's ability to support a secure virtual machine monitor", SSYM'00, pp. 10–10, 2000. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251306.1251316.

[24] VMware, *VMware products*, http://www.vmware.com/products/, [Online; accessed 22-September-2015], 2015.

[25] Linux, *Linux 2.6.20 change log*, "http://kernelnewbies.org/Linux_2_6_20", [Online; accessed 22-September-2015], 5 February, 2007.

[26] S. Grinberg and S. Weiss, "Architectural virtualization extensions: A systems perspective", *Computer Science Review*, vol. 6, no. 5-6, pp. 209–224, 2012, ISSN: 1574-0137. DOI: 10.1016/j.cosrev.2012.09.002. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1574013712000342.

[27] QEMU, *Qemu.org wiki, main page*, http://wiki.qemu.org/Main_Page, [Online; accessed 22-September-2015], 2015.

[28] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization", in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '03, Bolton Landing, NY, USA: ACM, 2003, pp. 164–177, ISBN: 1-58113-757-5. DOI: 10.1145/945445.945462.

[29] libvirt, *Libvirt.org main page*, http://libvirt.org, [online; accessed 22-September-2015], 2015.

[30] VMWare, *Understanding full virtualization, paravirtualization, and hardware assist*, http://www.vmware.com/files/pdf/VMware_paravirtualization.pdf, [Online; accessed 22-September-2015]; Revision 20070911, Item: WP-028-PRD-01-01, 2007.

[31] G. Wang and T. E. Ng, "The impact of virtualization on network performance of Amazon EC2 data center", in *INFOCOM, 2010 Proceedings IEEE*, IEEE, 2010, pp. 1–9.

[32] Intel and NASA, *NASA's flexible cloud fabric: Moving cluster applications to the cloud*, http://www.intel.com/content/dam/www/public/us/en/documents/case-studies/10-gigabit-ethernet-nasa-case-study.pdf, [Online; accessed 10-October-2015], 2011.

[33] PCI-SIG, *PCI-SIG SR-IOV specification*, http://pcisig.com/specifications/iov/ats, Login required, unable to access document, Jan. 2010.

[34] T. H. Romer, W. H. Ohlrich, A. R. Karlin, and B. N. Bershad, "Reducing TLB and memory overhead using online superpage promotion", *SIGARCH Comput. Archit. News*, vol. 23, no. 2, pp. 176–187, May 1995, ISSN: 0163-5964. DOI: 10.1145/225830.224419. [Online]. Available: http://doi.acm.org/10.1145/225830.224419.

[35] M. T. Jones, *Linux virtualization and PCI passthrough device emulation and hardware I/O virtualization*, http://www.ibm.com/developerworks/linux/library/l-pci-passthrough/l-pci-passthrough-pdf.pdf, [Online; accessed 22-September-2015], Oct. 2009.

[36] Linux-kvm.org, *Virtio paravirtualized drivers for Linux*, http://www.linux-kvm.org/page/Virtio, [Online, accessed 10-October-2015], Sep. 2015.

[37] Intel, *Intel virtualization technology for directed I/O*, http://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/vt-directed-io-spec.pdf, [Online whitepaper, retrieved 20-September-2015], Oct. 2014.

[38] softpanorama.com, *Slightly skeptical view on solaris zones*, http://softpanorama.org/Solaris/Virtualization/zones.shtml, [Online, accesses 15-October-2015], Oct. 2015.

[39] T. P. Morgan, *Google leverages container expertise on its cloud*, http://www.enterprisetech.com/2014/11/04/google-leverages-container-expertise-cloud/, [Online, accessed 15-October-2015], Nov. 2014.

[40] P. Menage, *Cgroups*, https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt, [Online, accessed 22-September-2015], May 2015.

[41] docker.com, *Network configuration*, https://docs.docker.com/articles/networking/, [Online; accessed 10-October-2015], Sep. 2015.

[42] J. Petazzoni, *Containers & Docker: How secure are they?*, http://blog.docker.com/2013/08/containers-docker-how-secure-are-they/, [Online, accessed 10-August-2015], Aug. 2013.

[43] Open Container Initiative, *Open container initiative*, https://www.opencontainers.org/pressrelease/, [Online; accessed 13-Octber-2015], Jun. 2015.

[44] docker.com, *Docker - the open-source application container engine*, https://github.com/docker/docker/, [Online; accessed 22-September-2015], Sep. 2015.

[45] ——, *Understanding docker*, https://docs.docker.com/introduction/understanding-docker/, [Online; accessed 26-August-2015], Sep. 2015.

[46] J. Savill, *What is docker?*, http://windowsitpro.com/windows/what-docker, [Online; accessed 12-October-2015], Jan. 2015.

[47] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar, "Making middleboxes someone else's problem: Network processing as a cloud service", *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 13–24, 2012.

[48] V. Sekar, S. Ratnasamy, M. K. Reiter, N. Egi, and G. Shi, "The middlebox manifesto: Enabling innovation in middlebox deployment", in *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, ACM, 2011, p. 21.

[49] S.-H. Ha, D. L. Pacheco, and G. Urvoy-Keller, "Impact of virtualization on network performance: The TCP case", in *CLOUDNET 2013 IEEE International Conference on Cloud*, IEEE, Nov. 2013.

[50] *DPDK data plane development kit*, `http://dpdk.org`, Website accessed 22-September-2015, Sep. 2015.

[51] K. Adams, "A comparison of software and hardware techniques for x86 virtualization", in *ASPLOS-XII: Proceedings of the 12th international conference on Architectural*, ACM Press, 2006, pp. 2–13.

[52] N. M. K. Chowdhury and R. Boutaba, "A survey of network virtualization", *Comput. Netw.*, vol. 54, no. 5, pp. 862–876, Apr. 2010, ISSN: 1389-1286. DOI: `10.1016/j.comnet.2009.10.017`.

[53] A. Kivity, D. Laor, G. Costa, P. Enberg, N. Har'El, D. Marti, and V. Zolotarov, "OSv–optimizing the operating system for virtual machines", in *2014 USENIX annual technical conference (USENIX ATC 14)*, USENIX Association, vol. 1, 2014, pp. 61–72.

[54] G. Wang and T. E. Ng, "Understanding network communication performance in virtualized cloud", *IEEE Multimedia Communication Technical Committee E-Letter*, 2011.

[55] J. Petazzoni, *Jpetazzo/pipework*, `https://github.com/jpetazzo/pipework`, [github repo; accessed 22-September-2015], 2015.

[56] J. Anderson, *Rakurai/pipework*, `https://github.com/Rakurai/pipework`, [github repo; accessed 22-September-2015], 2015.

[57] N. Marais, *Assign physical interface to docker exclusively*, `http://serverfault.com/questions/688483/assign-physical-interface-to-docker-exclusively`, [serverfault.com post; accessed 26-August-2015], Jun. 2015.

[58] J. Petazzoni, *Deep dive into docker storage drivers*, `http://jpetazzo.github.io/assets/2015-03-03-not-so-deep-dive-into-docker-storage-drivers.html`, [Online; accessed 15-October-2015], Mar. 2015.

[59] J. Eder, *Comprehensive overview of storage scalability in docker*, http://developerblog.redhat.com/2014/09/30/overview-storage-scalability-docker/, [Online; accessed 26-August-2015], Sep. 2014.

[60] Linux, *Index of /pub/linux/kernel/projects/rt/3.18/older*, https://www.kernel.org/pub/linux/kernel/projects/rt/3.18/older/, [Linux kernel patchset, downloaded 27-August-2015], Aug. 2015.

[61] L. maintainers, *Kernel parameters*, https://www.kernel.org/doc/Documentation/kernel-parameters.txt, [Online, accessed 2-August-2015], 2015.

[62] docker.com, *Dockerfile reference*, https://docs.docker.com/reference/builder/, [Online; accessed 10-October-2015], Sep. 2015.

[63] ——, *Best practices for writing Dockerfiles*, https://docs.docker.com/articles/dockerfile_best-practices/, [Online; accessed 10-October-2015], Sep. 2015.

[64] netperf.org, *The netperf homepage*, http://www.netperf.org/netperf, [Online; accessed 12-October-2015], Jul. 2015.

[65] ——, *Care and feeding of netperf 2.6.x*, http://www.netperf.org/svn/netperf2/tags/netperf-2.7.0/doc/netperf.html, [Online; accessed 12-October-2015], Jun. 2014.

[66] ——, *Netperf 2.7.0 source code*, ftp://ftp.netperf.org/netperf/netperf-2.7.0.tar.bz2, [netperf source code; downloaded 10-August-2015], Jul. 2015.

[67] L. LaValley, *Error in running netperf udp stream over openvpn*, http://stackoverflow.com/questions/11981480/error-in-running-netperf-udp-stream-over-openvpn, [stackoverflow.com post; accessed 20-October-2015], Jun. 2014.

[68] esnet, *Iperf3: a TCP, UDP, and SCTP network bandwidth measurement tool*, https://github.com/esnet/iperf, [Online github repo; retreived 6-September-2015], Sep. 2015.

[69] HP-IND Networking Performance Team, *Netperf manual: sec 5: using netperf to measure request/response performance*, http://www.netperf.org/netperf/training/Netperf.html, [Online; accessed 25-October-2015], Jul. 1995.